



Kea Administrator Reference Manual





Contents

1	Introduction	1
1.1	Supported Platforms	1
1.2	Required Software at Run-time	1
1.3	Kea Software	2
2	Quick Start	3
2.1	Quick Start Guide for DHCPv4 and DHCPv6 Services	3
2.2	Running the Kea Servers Directly	4
3	Installation	5
3.1	Packages	5
3.2	Installation Hierarchy	5
3.3	Building Requirements	5
3.4	Installation from Source	6
3.4.1	Download Tar File	6
3.4.2	Retrieve from Git	7
3.4.3	Configure Before the Build	7
3.4.4	Build	9
3.4.5	Install	9
3.5	DHCP Database Installation and Configuration	10
3.5.1	Building with MySQL Support	10
3.5.2	Building with PostgreSQL support	10
3.5.3	Building with CQL (Cassandra) Support	10
3.6	Hammer Building Tool	11
4	Kea Database Administration	13
4.1	Databases and Database Version Numbers	13
4.2	The kea-admin Tool	13
4.3	Supported Backends	14
4.3.1	memfile	14
4.3.1.1	Upgrading Memfile Lease Files from an Earlier Version of Kea	14



4.3.2	MySQL	15
4.3.2.1	First-Time Creation of the MySQL Database	15
4.3.2.2	Upgrading a MySQL Database from an Earlier Version of Kea	16
4.3.3	PostgreSQL	16
4.3.3.1	First-Time Creation of the PostgreSQL Database	16
4.3.3.2	Initialize the PostgreSQL Database Using kea-admin	17
4.3.3.3	Upgrading a PostgreSQL Database from an Earlier Version of Kea	17
4.3.4	Cassandra	18
4.3.4.1	First-Time Creation of the Cassandra Database	18
4.3.4.2	Upgrading a Cassandra Database from an Earlier Version of Kea	19
4.3.5	Using Read-Only Databases with Host Reservations	19
4.3.6	Limitations Related to the Use of SQL Databases	19
4.3.6.1	Year 2038 Issue	19
5	Kea Configuration	20
5.1	JSON Configuration	20
5.1.1	JSON Syntax	20
5.1.2	Simplified Notation	21
5.2	Kea Configuration Backend	21
5.2.1	Applicability	21
5.2.2	CB Capabilities and Limitations	22
5.2.3	CB Components	23
5.2.4	Configuration Sharing and Server Tags	23
6	Managing Kea with keactrl	25
6.1	Overview	25
6.2	Command Line Options	25
6.3	The keactrl Configuration File	25
6.4	Commands	26
6.5	Overriding the Server Selection	28
7	The Kea Control Agent	30
7.1	Overview	30
7.2	Configuration	30
7.3	Secure Connections	32
7.4	Starting the Control Agent	33
7.5	Connecting to the Control Agent	33



8	The DHCPv4 Server	34
8.1	Starting and Stopping the DHCPv4 Server	34
8.2	DHCPv4 Server Configuration	35
8.2.1	Introduction	35
8.2.2	Lease Storage	37
8.2.2.1	Memfile - Basic Storage for Leases	37
8.2.2.2	Lease Database Configuration	38
8.2.2.3	Cassandra-Specific Parameters	40
8.2.3	Hosts Storage	40
8.2.3.1	DHCPv4 Hosts Database Configuration	41
8.2.3.2	Using Read-Only Databases for Host Reservations	42
8.2.4	Interface Configuration	42
8.2.5	Issues with Unicast Responses to DHCPINFORM	44
8.2.6	IPv4 Subnet Identifier	45
8.2.7	IPv4 Subnet Prefix	45
8.2.8	Configuration of IPv4 Address Pools	46
8.2.9	Sending T1 (Option 58) and T2 (Option 59)	47
8.2.10	Standard DHCPv4 Options	48
8.2.11	Custom DHCPv4 Options	51
8.2.12	DHCPv4 Private Options	55
8.2.13	DHCPv4 Vendor-Specific Options	57
8.2.14	Nested DHCPv4 Options (Custom Option Spaces)	60
8.2.15	Unspecified Parameters for DHCPv4 Option Configuration	61
8.2.16	Stateless Configuration of DHCPv4 Clients	61
8.2.17	Client Classification in DHCPv4	62
8.2.17.1	Setting Fixed Fields in Classification	63
8.2.17.2	Using Vendor Class Information in Classification	63
8.2.17.3	Defining and Using Custom Classes	64
8.2.17.4	Required Classification	64
8.2.18	DDNS for DHCPv4	65
8.2.18.1	DHCP-DDNS Server Connectivity	66
8.2.18.2	When Does the kea-dhcp4 Server Generate a DDNS Request?	67
8.2.18.3	kea-dhcp4 Name Generation for DDNS Update Requests	68
8.2.18.4	Sanitizing Client Host Name and FQDN Names	69
8.2.19	Next Server (siaddr)	70
8.2.20	Echoing Client-ID (RFC 6842)	71
8.2.21	Using Client Identifier and Hardware Address	71
8.2.22	Authoritative DHCPv4 Server Behavior	72
8.2.23	DHCPv4-over-DHCPv6: DHCPv4 Side	73



8.2.24	Sanity Checks in DHCPv4	74
8.3	Host Reservation in DHCPv4	75
8.3.1	Address Reservation Types	76
8.3.2	Conflicts in DHCPv4 Reservations	76
8.3.3	Reserving a Hostname	77
8.3.4	Including Specific DHCPv4 Options in Reservations	78
8.3.5	Reserving Next Server, Server Hostname, and Boot File Name	78
8.3.6	Reserving Client Classes in DHCPv4	79
8.3.7	Storing Host Reservations in MySQL, PostgreSQL, or Cassandra	80
8.3.8	Fine-Tuning DHCPv4 Host Reservation	80
8.3.9	Global Reservations in DHCPv4	81
8.4	Shared Networks in DHCPv4	82
8.4.1	Local and Relayed Traffic in Shared Networks	85
8.4.2	Client Classification in Shared Networks	86
8.4.3	Host Reservations in Shared Networks	87
8.5	Server Identifier in DHCPv4	88
8.6	How the DHCPv4 Server Selects a Subnet for the Client	89
8.6.1	Using a Specific Relay Agent for a Subnet	89
8.6.2	Segregating IPv4 Clients in a Cable Network	90
8.7	Duplicate Addresses (DHCPDECLINE Support)	90
8.8	Statistics in the DHCPv4 Server	91
8.9	Management API for the DHCPv4 Server	91
8.10	User Contexts in IPv4	93
8.11	Supported DHCP Standards	94
8.12	DHCPv4 Server Limitations	94
8.13	Kea DHCPv4 Server Examples	95
8.14	Configuration Backend in DHCPv4	95
8.14.1	Supported Parameters	95
8.14.2	Enabling Configuration Backend	95
9	The DHCPv6 Server	99
9.1	Starting and Stopping the DHCPv6 Server	99
9.2	DHCPv6 Server Configuration	100
9.2.1	Introduction	100
9.2.2	Lease Storage	102
9.2.2.1	Memfile - Basic Storage for Leases	102
9.2.2.2	Lease Database Configuration	103
9.2.2.3	Cassandra-Specific Parameters	104
9.2.3	Hosts Storage	105



9.2.3.1	DHCPv6 Hosts Database Configuration	105
9.2.3.2	Using Read-Only Databases for Host Reservations	106
9.2.4	Interface Configuration	107
9.2.5	IPv6 Subnet Identifier	107
9.2.6	IPv6 Subnet Prefix	108
9.2.7	Unicast Traffic Support	108
9.2.8	Subnet and Address Pool	109
9.2.9	Subnet and Prefix Delegation Pools	110
9.2.10	Prefix Exclude Option	111
9.2.11	Standard DHCPv6 Options	111
9.2.12	Common Software46 Options	115
9.2.12.1	Software46 Container Options	115
9.2.12.2	S46 Rule Option	117
9.2.12.3	S46 BR Option	117
9.2.12.4	S46 DMR Option	117
9.2.12.5	S46 IPv4/IPv6 Address Binding option.	118
9.2.12.6	S46 Port Parameters	118
9.2.13	Custom DHCPv6 Options	118
9.2.14	DHCPv6 Vendor-Specific Options	120
9.2.15	Nested DHCPv6 Options (Custom Option Spaces)	121
9.2.16	Unspecified Parameters for DHCPv6 Option Configuration	122
9.2.17	Controlling the Values Sent for T1 and T2 Times	123
9.2.18	IPv6 Subnet Selection	124
9.2.19	Rapid Commit	124
9.2.20	DHCPv6 Relays	125
9.2.21	Relay-Supplied Options	125
9.2.22	Client Classification in DHCPv6	126
9.2.22.1	Defining and Using Custom Classes	127
9.2.22.2	Required Classification	127
9.2.23	DDNS for DHCPv6	128
9.2.23.1	DHCP-DDNS Server Connectivity	129
9.2.23.2	When Does the kea-dhcp6 Server Generate a DDNS Request?	130
9.2.23.3	kea-dhcp6 Name Generation for DDNS Update Requests	131
9.2.23.4	Sanitizing Client FQDN Names	133
9.2.24	DHCPv4-over-DHCPv6: DHCPv6 Side	133
9.2.25	Sanity Checks in DHCPv6	135
9.3	Host Reservation in DHCPv6	135
9.3.1	Address/Prefix Reservation Types	137
9.3.2	Conflicts in DHCPv6 Reservations	137



9.3.3	Reserving a Hostname	138
9.3.4	Including Specific DHCPv6 Options in Reservations	138
9.3.5	Reserving Client Classes in DHCPv6	139
9.3.6	Storing Host Reservations in MySQL, PostgreSQL, or Cassandra	140
9.3.7	Fine Tuning DHCPv6 Host Reservation	140
9.3.8	Global Reservations in DHCPv6	142
9.4	Shared Networks in DHCPv6	143
9.4.1	Local and Relayed Traffic in Shared Networks	145
9.4.2	Client Classification in Shared Networks	146
9.4.3	Host Reservations in Shared Networks	148
9.5	Server Identifier in DHCPv6	149
9.6	DHCPv6 data directory	151
9.7	Stateless DHCPv6 (Information-Request Message)	152
9.8	Support for RFC 7550 (now part of RFC 8415)	152
9.9	Using a Specific Relay Agent for a Subnet	153
9.10	Segregating IPv6 Clients in a Cable Network	153
9.11	MAC/Hardware Addresses in DHCPv6	154
9.12	Duplicate Addresses (DECLINE Support)	155
9.13	Statistics in the DHCPv6 Server	156
9.14	Management API for the DHCPv6 Server	156
9.15	User Contexts in IPv6	158
9.16	Supported DHCPv6 Standards	159
9.17	DHCPv6 Server Limitations	160
9.18	Kea DHCPv6 server examples	160
9.19	Configuration Backend in DHCPv6	160
9.19.1	Supported Parameters	160
9.19.2	Enabling Configuration Backend	161
10	Lease Expiration in DHCPv4 and DHCPv6	163
10.1	Lease Reclamation	163
10.2	Lease Reclamation Configuration Parameters	164
10.3	Configuring Lease Reclamation	164
10.4	Configuring Lease Affinity	166
10.5	Reclaiming Expired Leases with Command	166
11	Congestion Handling in DHCPv4 and DHCPv6	167
11.1	What is Congestion?	167
11.2	Configuring Congestion Handling	167



12 The DHCP-DDNS Server	169
12.1 Overview	169
12.1.1 DNS Server Selection	169
12.1.2 Conflict Resolution	169
12.1.3 Dual-Stack Environments	170
12.2 Starting and Stopping the DHCP-DDNS Server	170
12.3 Configuring the DHCP-DDNS Server	171
12.3.1 Global Server Parameters	171
12.3.2 Management API for the D2 Server	172
12.3.3 TSIG Key List	172
12.3.4 Forward DDNS	174
12.3.4.1 Adding Forward DDNS Domains	174
12.3.4.1.1 Adding Forward DNS Servers	174
12.3.5 Reverse DDNS	175
12.3.5.1 Adding Reverse DDNS Domains	175
12.3.5.1.1 Adding Reverse DNS Servers	176
12.3.6 User Contexts in DDNS	177
12.3.7 Example DHCP-DDNS Server Configuration	177
12.4 DHCP-DDNS Server Limitations	179
13 The LFC Process	180
13.1 Overview	180
13.2 Command-Line Options	180
14 Client Classification	181
14.1 Client Classification Overview	181
14.2 Builtin Client Classes	182
14.3 Using Expressions in Classification	183
14.3.1 Logical operators	185
14.3.2 Substring	186
14.3.3 Concat	186
14.3.4 Ifelse	186
14.3.5 Hexstring	186
14.4 Configuring Classes	186
14.5 Using Static Host Reservations In Classification	188
14.6 Configuring Subnets With Class Information	188
14.7 Configuring Pools With Class Information	189
14.8 Using Classes	190
14.9 Classes and Hooks	190
14.10 Debugging Expressions	190



15 Hooks Libraries	192
15.1 Introduction	192
15.2 Installing Hook packages	192
15.3 Configuring Hooks Libraries	194
15.4 Available Hooks Libraries	195
15.4.1 user_chk: Checking User Access	196
15.4.2 legal_log: Forensic Logging Hooks	197
15.4.2.1 Log File Naming	197
15.4.2.2 DHCPv4 Log Entries	197
15.4.2.3 DHCPv6 Log Entries	199
15.4.2.4 Configuring the Forensic Log Hooks	200
15.4.2.5 Database backend	201
15.4.3 flex_id: Flexible Identifiers for Host Reservations	202
15.4.4 host_cmds: Host Commands	204
15.4.4.1 The subnet-id parameter	205
15.4.4.2 reservation-add command	205
15.4.4.3 reservation-get command	207
15.4.4.4 reservation-get-all command	208
15.4.4.5 reservation-get-page command	209
15.4.4.6 reservation-del command	210
15.4.5 lease_cmds: Lease Commands	211
15.4.5.1 lease4-add, lease6-add Commands	212
15.4.5.1.1 lease6-bulk-apply	214
15.4.5.1.2 lease4-get, lease6-get Commands	216
15.4.5.1.3 lease4-get-all, lease6-get-all Commands	217
15.4.5.1.4 lease4-get-page, lease6-get-page Commands	218
15.4.5.1.5 lease4-del, lease6-del Commands	220
15.4.5.1.6 lease4-update, lease6-update Commands	220
15.4.5.1.7 lease4-wipe, lease6-wipe Commands	221
15.4.6 subnet_cmds: Subnet Commands	221
15.4.6.1 subnet4-list command	222
15.4.6.2 subnet6-list command	223
15.4.6.3 subnet4-get command	223
15.4.6.4 subnet6-get command	224
15.4.6.5 subnet4-add	224
15.4.6.6 subnet6-add	225
15.4.6.7 subnet4-update	226
15.4.6.8 subnet6-update	227
15.4.6.9 subnet4-del command	227



15.4.6.10	subnet6-del command	228
15.4.6.11	network4-list, network6-list commands	228
15.4.6.12	network4-get, network6-get commands	229
15.4.6.13	network4-add, network6-add commands	230
15.4.6.14	network4-del, network6-del commands	231
15.4.6.15	network4-subnet-add, network6-subnet-add commands	231
15.4.6.16	network4-subnet-del, network6-subnet-del commands	232
15.4.7	class_cmds: Class Commands	233
15.4.7.1	class-add Command	233
15.4.7.2	class-update Command	233
15.4.7.3	class-del Command	234
15.4.7.4	class-list Command	234
15.4.7.5	class-get Command	235
15.4.8	cb_cmds: Configuration Backend Commands	235
15.4.8.1	Commands Structure	236
15.4.8.2	Control Commands for DHCP Servers	236
15.4.8.2.1	Metadata	237
15.4.8.2.2	remote-server4-del, remote-server6-del commands	237
15.4.8.2.3	remote-server4-get, remote-server6-get commands	238
15.4.8.2.4	remote-server4-get-all, remote-server6-get-all commands	239
15.4.8.2.5	remote-server4-set, remote-server6-set commands	239
15.4.8.2.6	remote-global-parameter4-del, remote-global-parameter6-del commands	240
15.4.8.2.7	remote-global-parameter4-get, remote-global-parameter6-get commands	240
15.4.8.2.8	remote-global-parameter4-get-all, remote-global-parameter6-get-all commands	242
15.4.8.2.9	remote-global-parameter4-set	243
15.4.8.2.10	remote-network4-del, remote-network6-del commands	243
15.4.8.2.11	remote-network4-get, remote-network6-get commands	244
15.4.8.2.12	remote-network4-list, remote-network6-list commands	244
15.4.8.2.13	remote-network4-set	245
15.4.8.2.14	remote-option-def4-del, remote-option-def6-del commands	246
15.4.8.2.15	remote-option-def4-get, remote-option-def6-get commands	247
15.4.8.2.16	remote-option-def4-get-all, remote-option-def6-get-all commands	247
15.4.8.2.17	remote-option-def4-set, remote-option-def6-set commands	248
15.4.8.2.18	remote-option4-global-del, remote-option6-global-del commands	248
15.4.8.2.19	remote-option4-global-get, remote-option6-global-get commands	249
15.4.8.2.20	remote-option4-global-get-all, remote-option6-global-get-all commands	249
15.4.8.2.21	remote-option4-global-set, remote-option6-global-set command	250
15.4.8.2.22	remote-subnet4-del-by-id, remote-subnet6-del-by-id command	250
15.4.8.2.23	remote-subnet4-del-by-prefix, remote-subnet6-del-by-prefix commands	251



15.4.8.2.24	remote-subnet4-get-by-id, remote-subnet6-get-by-id commands	251
15.4.8.2.25	remote-subnet4-get-by-prefix, remote-subnet6-get-by-prefix command	251
15.4.8.2.26	remote-subnet4-list, remote-subnet6-list commands	252
15.4.8.2.27	remote-subnet4-set, remote-subnet6-set commands	253
15.4.9	ha: High Availability	254
15.4.9.1	Supported Configurations	255
15.4.9.2	Clocks on Active Servers	255
15.4.9.3	Server States	256
15.4.9.4	Scope Transition in a Partner-Down Case	257
15.4.9.5	Load-Balancing Configuration	258
15.4.9.6	Load Balancing with Advanced Classification	260
15.4.9.7	Hot-Standby Configuration	262
15.4.9.8	Lease Information Sharing	264
15.4.9.9	Controlling Lease-Page Size Limit	265
15.4.9.10	Discussion About Timeouts	265
15.4.9.11	Pausing HA State Machine	266
15.4.9.12	Control Agent Configuration	268
15.4.9.13	Control Commands for High Availability	269
15.4.9.13.1	ha-sync Command	269
15.4.9.13.2	ha-scopes Command	269
15.4.9.13.3	ha-continue Command	270
15.4.10	stat_cmds: Supplemental Statistics Commands	270
15.4.10.1	stat-lease4-get, stat-lease6-get Commands	271
15.4.11	radius: RADIUS Server Support	273
15.4.11.1	Compilation and Installation of the RADIUS Hook	273
15.4.11.2	RADIUS Hook Configuration	277
15.4.12	host_cache: Caching Host Reservations	280
15.4.12.1	cache-flush Command	280
15.4.12.2	cache-clear Command	281
15.4.12.3	cache-size Command	281
15.4.12.4	cache-write Command	281
15.4.12.5	cache-load Command	281
15.4.12.6	cache-get Command	282
15.4.12.7	cache-get-by-id Command	282
15.4.12.8	cache-insert Command	282
15.4.12.9	cache-remove Command	283
15.5	User contexts	283



16 Statistics	285
16.1 Statistics Overview	285
16.2 Statistics Lifecycle	285
16.3 Commands for Manipulating Statistics	286
16.3.1 statistic-get Command	286
16.3.2 statistic-reset Command	286
16.3.3 statistic-remove Command	287
16.3.4 statistic-get-all Command	287
16.3.5 statistic-reset-all Command	287
16.3.6 statistic-remove-all Command	287
17 Management API	288
17.1 Data Syntax	288
17.2 Using the Control Channel	290
17.3 Commands Supported by Both the DHCPv4 and DHCPv6 Servers	290
17.3.1 build-report	290
17.3.2 config-get	290
17.3.3 config-reload	291
17.3.4 config-test	291
17.3.5 config-write	292
17.3.6 leases-reclaim	292
17.3.7 libreload	292
17.3.8 list-commands	292
17.3.9 config-set	293
17.3.10 shutdown	293
17.3.11 dhcp-disable	293
17.3.12 dhcp-enable	294
17.3.13 version-get	294
17.4 Commands Supported by D2 Server	294
17.5 Commands Supported by Control Agent	294
18 Logging	296
18.1 Logging Configuration	296
18.1.1 Loggers	296
18.1.1.1 name (string)	296
18.1.1.2 severity (string)	297
18.1.1.3 debuglevel (integer)	299
18.1.1.4 output_options (list)	299
18.1.1.4.1 output (string)	299



18.1.1.4.2	flush (true of false)	299
18.1.1.4.3	maxsize (integer)	299
18.1.1.4.4	maxver (integer)	299
18.1.1.5	Example Logger Configurations	300
18.1.2	Logging Message Format	300
18.1.3	Logging During Kea Startup	301
19	The Kea Shell	302
19.1	Overview	302
19.2	Shell Usage	302
20	YANG/NETCONF Support	304
20.1	Overview	304
20.2	Installing NETCONF	304
20.2.1	Installing NETCONF on Ubuntu 18.04	304
20.2.2	Installing NETCONF on CentOS 7.5	304
20.3	Quick Sysrepo Overview	304
20.4	Supported YANG Models	307
20.5	Using the NETCONF Agent	307
20.6	Configuration	307
20.7	kea-netconf Configuration Example	309
20.8	Starting and Stopping the NETCONF Agent	311
20.9	Step-by-Step NETCONF Agent Operation Example	311
20.9.1	Setup of NETCONF Agent Operation Example	312
20.9.2	Error Handling in NETCONF Operation Example	313
20.9.3	NETCONF Operation Example with Two Pools	315
20.9.4	NETCONF Operation Example with Two Subnets	315
20.9.5	NETCONF Operation Example with Logging	316
A	API Reference	318
A.1	build-report reference	319
A.2	cache-clear reference	320
A.3	cache-get reference	320
A.4	cache-get-by-id reference	321
A.5	cache-insert reference	322
A.6	cache-load reference	323
A.7	cache-remove reference	323
A.8	cache-size reference	324
A.9	cache-write reference	325
A.10	class-add reference	325



A.11 class-del reference	326
A.12 class-get reference	326
A.13 class-list reference	327
A.14 class-update reference	328
A.15 config-get reference	328
A.16 config-reload reference	329
A.17 config-set reference	330
A.18 config-test reference	330
A.19 config-write reference	331
A.20 dhcp-disable reference	332
A.21 dhcp-enable reference	332
A.22 ha-continue reference	333
A.23 ha-heartbeat reference	333
A.24 ha-scopes reference	334
A.25 ha-sync reference	334
A.26 lease4-add reference	335
A.27 lease4-del reference	336
A.28 lease4-get reference	336
A.29 lease4-get-all reference	337
A.30 lease4-update reference	338
A.31 lease4-wipe reference	338
A.32 lease6-add reference	339
A.33 lease6-bulk-apply reference	340
A.34 lease6-del reference	341
A.35 lease6-get reference	342
A.36 lease6-get-all reference	342
A.37 lease6-update reference	343
A.38 lease6-wipe reference	344
A.39 leases-reclaim reference	345
A.40 libreload reference	345
A.41 list-commands reference	346
A.42 network4-add reference	346
A.43 network4-del reference	347
A.44 network4-get reference	348
A.45 network4-list reference	349
A.46 network4-subnet-add reference	350
A.47 network4-subnet-del reference	350
A.48 network6-add reference	351
A.49 network6-del reference	352



A.50 network6-get reference	353
A.51 network6-list reference	353
A.52 network6-subnet-add reference	354
A.53 network6-subnet-del reference	355
A.54 remote-global-parameter4-del reference	355
A.55 remote-global-parameter4-get reference	356
A.56 remote-global-parameter4-get-all reference	357
A.57 remote-global-parameter4-set reference	358
A.58 remote-global-parameter6-del reference	359
A.59 remote-global-parameter6-get reference	360
A.60 remote-global-parameter6-get-all reference	360
A.61 remote-global-parameter6-set reference	361
A.62 remote-network4-del reference	362
A.63 remote-network4-get reference	363
A.64 remote-network4-list reference	364
A.65 remote-network4-set reference	365
A.66 remote-network6-del reference	366
A.67 remote-network6-get reference	366
A.68 remote-network6-list reference	367
A.69 remote-network6-set reference	368
A.70 remote-option-def4-del reference	369
A.71 remote-option-def4-get reference	370
A.72 remote-option-def4-get-all reference	371
A.73 remote-option-def4-set reference	372
A.74 remote-option-def6-del reference	372
A.75 remote-option-def6-get reference	373
A.76 remote-option-def6-get-all reference	374
A.77 remote-option-def6-set reference	375
A.78 remote-option4-global-del reference	376
A.79 remote-option4-global-get reference	377
A.80 remote-option4-global-get-all reference	377
A.81 remote-option4-global-set reference	378
A.82 remote-option6-global-del reference	379
A.83 remote-option6-global-get reference	380
A.84 remote-option6-global-get-all reference	381
A.85 remote-option6-global-set reference	382
A.86 remote-server4-del reference	383
A.87 remote-server4-get reference	384
A.88 remote-server4-get-all reference	384



A.89 remote-server4-set reference	385
A.90 remote-server6-del reference	386
A.91 remote-server6-get reference	387
A.92 remote-server6-get-all reference	388
A.93 remote-server6-set reference	388
A.94 remote-subnet4-del-by-id reference	389
A.95 remote-subnet4-del-by-prefix reference	390
A.96 remote-subnet4-get-by-id reference	391
A.97 remote-subnet4-get-by-prefix reference	392
A.98 remote-subnet4-list reference	393
A.99 remote-subnet4-set reference	394
A.100remote-subnet6-del-by-id reference	394
A.101remote-subnet6-del-by-prefix reference	395
A.102remote-subnet6-get-by-id reference	396
A.103remote-subnet6-get-by-prefix reference	397
A.104remote-subnet6-list reference	398
A.105remote-subnet6-set reference	399
A.106reservation-add reference	400
A.107reservation-del reference	401
A.108reservation-get reference	401
A.109reservation-get-all reference	402
A.110reservation-get-page reference	403
A.111shutdown reference	403
A.112stat-lease4-get reference	404
A.113stat-lease6-get reference	405
A.114statistic-get reference	406
A.115statistic-get-all reference	406
A.116statistic-remove reference	407
A.117statistic-remove-all reference	408
A.118statistic-reset reference	408
A.119statistic-reset-all reference	409
A.120subnet4-add reference	410
A.121subnet4-del reference	410
A.122subnet4-get reference	411
A.123subnet4-list reference	412
A.124subnet4-update reference	413
A.125subnet6-add reference	413
A.126subnet6-del reference	414
A.127subnet6-get reference	415
A.128subnet6-list reference	416
A.129subnet6-update reference	416
A.130version-get reference	417



B Acknowledgments



List of Tables

4.1	List of available backends	14
8.1	List of Standard DHCPv4 Options	52
8.2	List of Standard DHCP Option Types	53
8.3	Default FQDN Flag Behavior	67
8.4	DHCPv4 Statistics	92
8.5	List of DHCPv4 Parameters Supported by the Configuration Backend	96
9.1	List of Standard DHCPv6 Options	116
9.2	Default FQDN Flag Behavior	131
9.3	DHCPv6 Statistics	157
9.4	List of DHCPv6 Parameters Supported by the Configuration Backend	161
12.1	Our Example Network	177
12.2	Forward DDNS Domains Needed	177
12.3	Reverse DDNS Domains Needed	178
14.1	List of Classification Values	184
14.2	List of Classification Expressions	185
15.1	List of available hooks libraries	195
15.2	Default Behavior of the Server in Various HA States	257
18.1	List of loggers supported by Kea servers and hooks libraries shipped with Kea and premium packages	298

Abstract

Kea is an open source implementation of the Dynamic Host Configuration Protocol (DHCP) servers, developed and maintained by Internet Systems Consortium (ISC).

This is the reference guide for Kea version 1.6.0-beta2. Links to the most up-to-date version of this document (in PDF, HTML, and plain text formats), along with other documents for Kea, can be found in ISC's [Knowledgebase](#).



Chapter 1

Introduction

Kea is the next generation of DHCP software developed by ISC. It supports both DHCPv4 and DHCPv6 protocols along with their extensions, e.g. prefix delegation and dynamic updates to DNS.

This guide covers Kea version 1.6.0-beta2.

1.1 Supported Platforms

Kea is officially supported on CentOS, Fedora, Ubuntu, Debian, and FreeBSD systems. It is also likely to work on many other platforms. Kea-1.6.0-beta2 builds have been tested on:

- CentOS Linux — 7.1804 (aka 7.5)
- Fedora — 28, 29
- Ubuntu — 16.04, 18.04
- Debian GNU/Linux — 8, 9
- FreeBSD — 11.0
- macOS — 10.13, 10.14

There are currently no plans to port Kea to Windows platforms.

1.2 Required Software at Run-time

Running Kea uses various extra software packages which may not be provided in the default installation of some operating systems, nor in the standard package collections. You may need to install this required software separately. (For the build requirements, also see Section 3.3.)

- Kea supports two cryptographic libraries: Botan and OpenSSL. Only one of them is required to be installed during compilation. Kea uses the Botan library for C++ (<http://botan.randombit.net/>), version 1.9 or later. Note that support for Botan versions earlier than 2.0 will be removed in Kea 1.6.0 and later. As an alternative to Botan, Kea can use the OpenSSL cryptographic library (<http://www.openssl.org/>), version 1.0.2 or later.
 - Kea uses the log4cplus C++ logging library (<http://log4cplus.sourceforge.net/>). It requires log4cplus version 1.0.3 or later.
 - Kea requires the Boost system library (<http://www.boost.org/>). Building with the header-only version of Boost is no longer recommended.
-



- In order to store lease information in a MySQL database, Kea requires MySQL headers and libraries. This is an optional dependency in that Kea can be built without MySQL support.
- In order to store lease information in a PostgreSQL database, Kea requires PostgreSQL headers and libraries. This is an optional dependency in that Kea can be built without PostgreSQL support.
- In order to store lease information in a Cassandra database (CQL), Kea requires Cassandra headers and libraries. This is an optional dependency in that Kea can be built without Cassandra support.
- Integration with RADIUS is provided in Kea via the hooks library available to our paid support customers. Use of this library requires the FreeRadius-client library to be present on the system where Kea is running. This is an optional dependency in that Kea can be built without RADIUS support.
- As of the 1.5.0 release, Kea provides a NETCONF interface with the kea-netconf agent. This Kea module is built optionally and requires Sysrepo software when used. Building Kea with NETCONF support requires many dependencies to be installed, which are described in more detail in Section [20.2](#).

1.3 Kea Software

Kea is modular. Part of this modularity is accomplished using multiple cooperating processes which, together, provide the server functionality. The following software is included with Kea:

- **keactrl** — Tool to start, stop, reconfigure, and report status for the Kea servers.
- **kea-dhcp4** — The DHCPv4 server process. This process responds to DHCPv4 queries from clients.
- **kea-dhcp6** — The DHCPv6 server process. This process responds to DHCPv6 queries from clients.
- **kea-dhcp-ddns** — The DHCP Dynamic DNS process. This process acts as an intermediary between the DHCP servers and DNS servers. It receives name update requests from the DHCP servers and sends DNS update messages to the DNS servers.
- **kea-admin** — A useful tool for database backend maintenance (creating a new database, checking versions, upgrading, etc.)
- **kea-lfc** — This process removes redundant information from the files used to provide persistent storage for the memfile database backend. While it can be run standalone, it is normally run as and when required by the Kea DHCP servers.
- **kea-ctrl-agent** — Kea Control Agent (CA) is a daemon that exposes a RESTful control interface for managing Kea servers.
- **kea-netconf** - Kea netconf is an agent that provides a YANG/NETCONF interface for the Kea environment.
- **kea-shell** — This simple text client uses the REST interface to connect to the Kea Control Agent.
- **perfdhcp** — A DHCP benchmarking tool which simulates multiple clients to test both DHCPv4 and DHCPv6 server performance.

The tools and modules are covered in full detail in this guide. In addition, manual pages are also provided in the default installation.

Kea also provides C++ libraries and programmer interfaces for DHCP. These include detailed developer documentation and code examples.



Chapter 2

Quick Start

This section describes the basic steps needed to get Kea up and running. For further details, full customizations, and troubleshooting, see the respective chapters in the Kea Administrator Reference Manual (ARM).

2.1 Quick Start Guide for DHCPv4 and DHCPv6 Services

1. Install required run-time and build dependencies. See Section 3.3 for details.
2. Download the Kea source tarball from the [ISC.org downloads page](#) or the [ISC FTP server](#).
3. Extract the tarball. For example:

```
$ tar xvzf kea-1.6.0-beta2.tar.gz
```

4. Go into the source directory and run the configure script:

```
$ cd kea-1.6.0-beta2
$ ./configure [your extra parameters]
```

5. Build it:

```
$ make
```

6. Install it (by default it will be placed in `/usr/local/`, so it is likely that you will need root privileges for this step):

```
# make install
```

7. Edit the Kea configuration files which by default are installed in the `[kea-install-dir]/etc/kea/` directory. These are: `kea-dhcp4.conf`, `kea-dhcp6.conf`, `kea-dhcp-ddns.conf` and `kea-ctrl-agent.conf`, for DHCPv4 server, DHCPv6 server, D2, and Control Agent, respectively.

8. In order to start the DHCPv4 server in the background, run the following command (as root):

```
# keactrl start -s dhcp4
```

Or run the following command to start the DHCPv6 server instead:

```
# keactrl start -s dhcp6
```

Note that it is also possible to start all servers simultaneously:

```
$ keactrl start
```



9. Verify that the Kea server(s) is/are running:

```
# keactrl status
```

A server status of "inactive" may indicate a configuration error. Please check the log file (by default named [kea-install-dir]/var/log/kea-dhcp4.log, [kea-install-dir]/var/log/kea-dhcp6.log, [kea-install-dir]/var/log/kea-ddns.log or [kea-install-dir]/var/log/kea-ctrl-agent.log) for the details of the error.

10. If the server has been started successfully, test that it is responding to DHCP queries and that the client receives a configuration from the server; for example, use the [ISC DHCP client](#).
11. Stop running the server(s):

```
# keactrl stop
```

For instructions specific to your system, please read the [system-specific notes](#), available in the [Kea section of ISC's Knowledge-base](#).

The details of **keactrl** script usage can be found in [Chapter 6](#).

2.2 Running the Kea Servers Directly

The Kea servers can be started directly, without the need to use the **keactrl**. To start the DHCPv4 server run the following command:

```
# kea-dhcp4 -c /path/to/your/kea4/config/file.json
```

Similarly, to start the DHCPv6 server run the following command:

```
# kea-dhcp6 -c /path/to/your/kea6/config/file.json
```



Chapter 3

Installation

3.1 Packages

Some operating systems or software package vendors may provide ready-to-use, pre-built software packages for Kea. Installing a pre-built package means you do not need to install the software required only to build Kea and do not need to *make* the software.

3.2 Installation Hierarchy

The following is the directory layout of the complete Kea installation. (All directory paths are relative to the installation directory):

- `etc/kea/` — configuration files.
- `include/` — C++ development header files.
- `lib/` — libraries.
- `lib/kea/hooks` — additional hooks libraries.
- `sbin/` — server software and commands used by the system administrator.
- `share/kea/` — configuration specifications and examples.
- `share/doc/kea/` — this guide, other supplementary documentation, and examples.
- `share/man/` — manual pages (online documentation).
- `var/lib/kea/` — server identification, and lease databases files.
- `var/log/` — log files.
- `var/run/kea/` — pid and logger lock files.

3.3 Building Requirements

In addition to the run-time requirements (listed in Section 1.2), building Kea from source code requires various development include headers and program development tools.

Note

Some operating systems have split their distribution packages into a run-time and a development package. You will need to install the development package versions, which include header files and libraries, to build Kea from the source code.



Building from source code requires the following software installed on the system:

- Boost C++ Libraries (<http://www.boost.org/>). The oldest Boost version used for testing is 1.57 (although it may also work with older versions). The Boost system library must also be installed. Installing a header-only version of Boost is no longer recommended.
- OpenSSL (at least version 1.0.1) or Botan (at least version 1.9). Note that OpenSSL version 1.0.2 or 1.1.0 or later and Botan version 2 or later are strongly recommended.
- log4cplus (at least version 1.0.3) development include headers.
- A C++ compiler (with C++11 support) and standard development headers. Kea building was checked with GCC g++ 4.8.5 and some later versions and Clang 800.0.38 and some later versions.
- The development tools automake, libtool, pkg-config.
- The MySQL client and the client development libraries, when using the `--with-mysql` configuration flag to build the Kea MySQL database backend. In this case, an instance of the MySQL server running locally or on a machine reachable over a network is required. Note that running the unit tests requires a local MySQL server.
- The PostgreSQL client and the client development libraries, when using the `--with-pgsql` configuration flag to build the Kea PostgreSQL database backend. In this case an instance of the PostgreSQL server running locally or on some other machine, reachable over the network from the machine running Kea, is required. Note that running the unit tests requires a local PostgreSQL server.
- The cpp-driver from DataStax is needed when using the `--with-cql` configuration flag to build Kea with the Cassandra database backend. In this case, an instance of the Cassandra server running locally or on some other machine, reachable over the network from the machine running Kea, is required. Note that running the unit tests requires a local Cassandra server.
- The FreeRADIUS client library is required to connect to a RADIUS server. (This is specified using the `--with-freeradius` configuration switch.)
- Sysrepo (version 0.7.6 or later) and libyang (version 0.16-r2 or later) are needed to connect to a Sysrepo database. (This is specified using the `--with-sysrepo` switch when running "configure".)
- googletest (version 1.8 or later), when using the `--with-gtest` configuration option to build the unit tests.
- The documentation generation tools elinks, docbook-xsl, libxslt, and Doxygen, if using the `--enable-generate-docs` configuration option to create the documentation.

Visit ISC's Knowledgebase at <https://kb.isc.org/docs/installing-kea> for system-specific installation tips.

3.4 Installation from Source

Although Kea may be available in pre-compiled, ready-to-use packages from operating system vendors, it is open source software written in C++. As such, it is freely available in source code form from ISC as a downloadable tar file. The source code can also be obtained from the Kea Gitlab repository at (<https://gitlab.isc.org/isc-projects/kea>). This section describes how to build Kea from the source code.

3.4.1 Download Tar File

The Kea release tarballs may be downloaded from: <http://ftp.isc.org/isc/kea/> (using FTP or HTTP).



3.4.2 Retrieve from Git

Downloading this "bleeding edge" code is recommended only for developers or advanced users. Using development code in a production environment is not recommended.

Note

When building from source code retrieved via Git, additional software will be required: automake (v1.11 or later), libtoolize, and autoconf (v2.69 or later). These may need to be installed.

The latest development code is available on Gitlab (see <https://gitlab.isc.org/isc-projects/kea>). The Kea source is public and development is done in the "master" branch.

The code can be checked out from <https://gitlab.isc.org/isc-projects/kea.git>:

```
$ git clone https://gitlab.isc.org/isc-projects/kea.git
```

The code checked out from the git repository does not include the generated configure script, Makefile.in files, nor their related build files. They can be created by running **autoreconf** with the `--install` switch. This will run **autoconf**, **aclocal**, **libtoolize**, **autoheader**, **automake**, and related commands.

Write access to the Kea repository is only granted to ISC staff. If you are a developer planning to contribute to Kea, please check our [Contributor's Guide](#). The [Kea Developer's Guide](#) contains more information about the process, as well as describes the requirements for contributed code to be accepted by ISC.

3.4.3 Configure Before the Build

Kea uses the GNU Build System to discover build environment details. To generate the makefiles using the defaults, simply run:

```
$ ./configure
```

Run `./configure` with the `--help` switch to view the different options. Some commonly-used options are:

--prefix

Define the installation location (the default is `/usr/local`).

--with-mysql

Build Kea with code to allow it to store leases and host reservations in a MySQL database.

--with-pgsql

Build Kea with code to allow it to store leases and host reservations in a PostgreSQL database.

--with-cql

Build Kea with code to allow it to store leases and host reservations in a Cassandra (CQL) database.

--with-log4cplus

Define the path to find the Log4cplus headers and libraries. Normally this is not necessary.

--with-boost-include

Define the path to find the Boost headers. Normally this is not necessary.

--with-botan-config

Specify the path to the botan-config script to build with Botan for cryptographic functions. It is preferred to use OpenSSL (see below).

--with-openssl

Replace Botan by the OpenSSL the cryptographic library. By default **configure** searches for a valid Botan installation. If one is not found, it searches for OpenSSL. Normally this is not necessary.

**--enable-shell**

Build the optional **kea-shell** tool (more in Chapter 19). The default is to not build it.

--enable-perfdhcp

Build the optional **perfdhcp** DHCP benchmarking tool. The default is to not build it.

Note

The `--runstatedir` in the installation directories is particular: there are three cases:

1. You use autoconf 2.70 or greater which supports this but this autoconf version has not been currently released.
2. You use autoconf 2.69 patched to add support of this. In this case and the previous simply use when needed the `--runstatedir` configure parameter.
3. There is no support (the configure parameter is not recognized and configure directly raises an error). For autoconf 2.69 the `runstatedir` environment variable is supported so simply remove the `--` before `runstatedir` in the configure script call.

Note

For instructions concerning the installation and configuration of database backends for Kea, see Section 3.5.

There are also many additional options that are typically not used by regular users. However, they may be useful for package maintainers, developers or people who want to extend Kea code or send patches:

--with-gtest, --with-gtest-source

Enable the building of the C++ Unit Tests using the Google Test framework. This option specifies the path to the gtest source. (If the framework is not installed on your system, it can be downloaded from <https://github.com/google/googletest>.)

--enable-generate-docs

Enable the rebuilding Kea documentation. ISC publishes Kea documentation for each release. However, in some cases you may want to rebuild it. For example, if you want to change something in the docs, or want to generate new one from git sources that are not released yet. The build procedure uses the `xsltproc` tool with the `nonet` argument which disables fetching missing sources, e.g. `docbook.xsl`, from the Internet. If you want anyway to use the Internet please set the `XSLTPROC_NET` environment variable in `configure` to any non-empty value, e.g.

```
$ ./configure XSLTPROC_NET=yes --enable-generate-docs
```

--enable-generate-parser

Many Kea components have parsers implemented using flex (.ll files) and bison (.yy files). Kea sources have C++/h files generated out from them. By default Kea does not use flex or bison to avoid requiring installation of unnecessary dependencies for users. However, if you change anything in the parses (such as add new parameter), you will need to use flex and bison to regenerate parsers. This option lets you do that.

--enable-generate-messages

Enable the regeneration of messages files from their messages source files, e.g. regenerate `xxx_messages.h` and `xxx_messages.cc` from `xxx_messages.mes` using the Kea message compiler. By default Kea is built using these .h and .cc files from the distribution. However, if you change anything in a .mes file (such as add new message), you will need to build and use the Kea message compiler. This option lets you do that.

--with-benchmark, --with-benchmark-source

Enable the building of the database backend benchmarks using the Google Benchmark framework. This option specifies the path to the gtest source. (If the framework is not installed on your system, it can be downloaded from <https://github.com/google/benchmark>.) This support is experimental.



For example, the following command configures Kea to find the Boost headers in `/usr/pkg/include`, specifies that PostgreSQL support should be enabled, and sets the installation location to `/opt/kea`:

```
$ ./configure \  
    --with-boost-include=/usr/pkg/include \  
    --with-pgsql=/usr/local/bin/pg_config \  
    --prefix=/opt/kea
```

If you have any problems with building Kea using the header-only Boost code, or you'd like to use the Boost system library (assumed for the sake of this example to be located in `/usr/pkg/lib`):

```
$ ./configure \  
    --with-boost-libs=-lboost_system \  
    --with-boost-lib-dir=/usr/pkg/lib
```

If `configure` fails, it may be due to missing or old dependencies.

If `configure` succeeds, it displays a report with the parameters used to build the code. This report is saved into the file `config.report` and is also embedded into the executable binaries, e.g., `kea-dhcp4`.

3.4.4 Build

After the configure step is complete, build the executables from the C++ code and prepare the Python scripts by running the command:

```
$ make
```

3.4.5 Install

To install the Kea executables, support files, and documentation, issue the command:

```
$ make install
```

Do not use any form of parallel or job server options (such as GNU make's `-j` option) when performing this step; doing so may cause errors.

Note

The install step may require superuser privileges.

If required, run `ldconfig` as root with `/usr/local/lib` (or with `prefix/lib` if configured with `--prefix`) in `/etc/ld.so.conf` (or the relevant linker cache configuration file for your OS):

```
$ ldconfig
```

Note

If you do not run `ldconfig` where it is required, you may see errors like the following:

```
program: error while loading shared libraries: libkea-something.so.1:  
cannot open shared object file: No such file or directory
```



3.5 DHCP Database Installation and Configuration

Kea stores its leases in a lease database. The software has been written in a way that makes it possible to choose which database product should be used to store the lease information. At present, Kea supports four database backends: MySQL, PostgreSQL, Cassandra, and Memfile. To limit external dependencies, MySQL, PostgreSQL, and Cassandra support are disabled by default and only Memfile is available. Support for the optional external database backend must be explicitly included when Kea is built. This section covers the building of Kea with one of the optional backends and the creation of the lease database.

Note

When unit tests are built with Kea (the `--with-gtest` configuration option is specified), the databases must be manually pre-configured for the unit tests to run. The details of this configuration can be found in the [Kea Developer's Guide](#).

3.5.1 Building with MySQL Support

Install MySQL according to the instructions for your system. The client development libraries must be installed.

Build and install Kea as described in Chapter 3, with the following modification. To enable the MySQL database code, at the "configure" step (see Section 3.4.3), the `--with-mysql` switch should be specified:

```
./configure [other-options] --with-mysql
```

If MySQL was not installed in the default location, the location of the MySQL configuration program "mysql_config" should be included with the switch, i.e.

```
./configure [other-options] --with-mysql=path-to-mysql_config
```

See Section 4.3.2.1 for details regarding MySQL database configuration.

3.5.2 Building with PostgreSQL support

Install PostgreSQL according to the instructions for your system. The client development libraries must be installed. Client development libraries are often packaged as "libpq".

Build and install Kea as described in Chapter 3, with the following modification. To enable the PostgreSQL database code, at the "configure" step (see Section 3.4.3), the `--with-pgsql` switch should be specified:

```
./configure [other-options] --with-pgsql
```

If PostgreSQL was not installed in the default location, the location of the PostgreSQL configuration program "pg_config" should be included with the switch, i.e.

```
./configure [other-options] --with-pgsql=path-to-pg_config
```

See Section 4.3.3.1 for details regarding PostgreSQL database configuration.

3.5.3 Building with CQL (Cassandra) Support

Install Cassandra according to the instructions for your system. The Cassandra project website contains useful pointers: <http://cassandra.apache.org>.

If you have a `cpp-driver` package available as binary or as source, simply install or build and install the package. Then build and install Kea as described in Chapter 3. To enable the Cassandra (CQL) database code, at the "configure" step (see Section 3.4.3), do:

```
./configure [other-options] --with-cql=path-to-pkg_config
```



Note if **pkg-config** is at its standard location (and thus in the shell path) you do not need to supply its path. If it does not work (e.g. no **pkg-config**, package not available in **pkg-config** with the **cassandra** name), you can still use the **cql_config** script in **tools/** as described below.

Download and compile **cpp-driver** from DataStax. For details regarding dependencies for building **cpp-driver**, see the project homepage <https://github.com/datastax/cpp-driver>. In June 2016, the following commands were used:

```
$ git clone https://github.com/datastax/cpp-driver
$ cd cpp-driver
$ mkdir build
$ cd build
$ cmake ..
$ make
```

As of January 2019, **cpp-driver** does not include **cql_config** script. Work is in progress to contribute such a script to the **cpp-driver** project but, until that is complete, intermediate steps need to be conducted. A **cql_config** script is present in the **tools/** directory of the Kea sources. Before using it, please create a **cql_config_defines.sh** in the same directory (there is an example in **cql_config_define.sh.sample** available; you may copy it over to **cql_config_defines.sh** and edit the path specified in it) and change the environment variable **CPP_DRIVER_PATH** to point to the directory where the **cpp-driver** sources are located. Make sure that appropriate access rights are set on this file. It should be executable by the system user building Kea.

Build and install Kea as described in Chapter 3, with the following modification. To enable the Cassandra (CQL) database code, at the "configure" step (see Section 3.4.3), do:

```
./configure [other-options] --with-cql=path-to-cql_config
```

3.6 Hammer Building Tool

An optional building tool called Hammer was introduced with Kea 1.6.0. It is a Python 3 script that lets users automate tasks related to building Kea, such as setting up virtual machines, installing Kea dependencies, compiling Kea with various options, running unit-tests and more. This tool was created primarily for internal QA purposes at ISC and it is not included in the Kea distribution. However, it is available in the Kea git repository. This tool was developed primarily for internal purpose and ISC cannot guarantee its proper operation. If you decide to use it, please do so with care.

Note

Use of this tool is completely optional. Everything it does can be done manually.

The first time user is strongly encouraged to look at Hammer's built in help:

```
./hammer.py --help
```

It will list available parameters.

Hammer is able to set up various operating systems running either in LXC or in VirtualBox. To list of supported systems, use **supported-systems** command:

```
$/hammer.py supported-systems
fedora:
- 27: lxc, virtualbox
- 28: lxc, virtualbox
- 29: lxc, virtualbox
centos:
- 7: lxc, virtualbox
rhel:
- 8: virtualbox
ubuntu:
- 16.04: lxc, virtualbox
```



```
- 18.04: lxc, virtualbox
- 18.10: lxc, virtualbox
debian:
- 8: lxc, virtualbox
- 9: lxc, virtualbox
freebsd:
- 11.2: virtualbox
- 12.0: virtualbox
```

It is also possible to run build locally, in current system (if the OS is supported).

At first it is required to install Hammer dependencies which is Vagrant and either VirtualBox or LXC. To make life easier Hammer can install Vagrant and required Vagrant plugins using the command:

```
./hammer.py ensure-hammer-deps
```

VirtualBox and LXC need to be installed manually.

Basic functionality provided by Hammer is preparing building environment and performing actual build and running unit tests locally, in current system. This can be achieved by running the command:

```
./hammer.py build -p local
```

The scope of the process can be defined using `--with (-w)` and `--without (-x)` options. By default the build command will build Kea with documentation, install it locally and run unit tests.

To exclude installation and generating docs do:

```
./hammer.py build -p local -x install docs
```

The basic scope can be extended by: `mysql`, `pgsql`, `cql`, `native-pkg`, `radius`, `shell`, `forge`.

Note

For build Kea locally installing Hammer dependencies like Vagrant is not needed.

Hammer can be told to set up a new virtual machine with specified operating system and not running the build:

```
./hammer.py prepare-system -p virtualbox -s freebsd -r 12.0
```

This way we can prepare a system for our own use. To get to such system using SSH invoke:

```
./hammer.py ssh -p virtualbox -s freebsd -r 12.0
```

It is possible to speed up subsequent Hammer builds. To achieve this Hammer employs **ccache**. During compilation ccache stores object to shared folder. In subsequent runs instead doing actual compilation ccache just returns stored earlier objects. Cache with objects for reuse needs to be stored outside of VM or LXC. To indicate such folder Hammer requires providing `--ccache-dir` parameter. In indicated folder there are stored objects for each target operating system separately.

```
./hammer.py build -p lxc -s ubuntu -r 18.04 --ccache-dir ~/kea-ccache
</b>
```

Note

For now ccache is only supported for LXC provider in Hammer. Support for VirtualBox will be added later.

For more information check:

```
./hammer.py --help
```



Chapter 4

Kea Database Administration

4.1 Databases and Database Version Numbers

Kea may be configured to use a database as a storage for leases, a source of servers' configurations and host reservations (i.e. static assignments of addresses, prefixes, options etc.). Subsequent Kea releases introduce changes to the database schemas to facilitate new features and correct discovered issues with the existing schemas.

A given version of Kea expects a particular structure in the backend and checks for this by examining the version of database it is using. Separate version numbers are maintained for backends, independent of the version of Kea itself. It is possible that the backend version will stay the same through several Kea revisions; similarly, it is possible that the version of the backend may go up several revisions during a Kea upgrade. Versions for each backend are independent, so an increment in the MySQL backend version does not imply an increment in that of PostgreSQL.

Backend versions are specified in a *major.minor* format. The minor number is increased when there are backward-compatible changes introduced; for example, the addition of a new index. It is desirable but not mandatory to apply such a change; you can run an older backend version if you want to. (Although, in the example given, running without the new index may be at the expense of a performance penalty.) On the other hand, the major number is increased when an incompatible change is introduced: for example, an extra column is added to a table. If you try to run Kea on a backend that is too old (as signified by a mismatched backend major version number), Kea will refuse to run; administrative action will be required to upgrade the backend.

4.2 The kea-admin Tool

To manage the databases, Kea provides the **kea-admin** tool. It is able to initialize a new backend, check its version number, perform a backend upgrade, and dump lease data to a text file.

kea-admin takes two mandatory parameters: **command** and **backend**. Additional, non-mandatory options may be specified. The currently supported commands are:

- **db-init** — Initializes a new database schema. This is useful during a new Kea installation. The database is initialized to the latest version supported by the version of the software being installed.
- **db-version** — Reports the database backend version number. This is not necessarily equal to the Kea version number as each backend has its own versioning scheme.
- **db-upgrade** — Conducts a database schema upgrade. This is useful when upgrading Kea.
- **lease-dump** — Dumps the contents of the lease database (for MySQL, PostgreSQL, or CQL backends) to a CSV (comma-separated values) text file. The first line of the file contains the column names. This is meant to be used as a diagnostic tool, so it provides a portable, human-readable form of the lease data.



Note In previous versions of Kea ending with 1.6.0 **db-init**, **db-version** and **db-upgrade** commands were named **lease-init**, **lease-version** and **lease-upgrade**.

backend specifies the type of backend database. The currently supported types are:

- **memfile** — Lease information is stored on disk in a text file.
- **mysql** — Information is stored in a MySQL relational database.
- **pgsql** — Information is stored in a PostgreSQL relational database.
- **cql** — Information is stored in an Apache Cassandra database.

Additional parameters may be needed, depending on your setup and specific operation: username, password, and database name or the directory where specific files are located. See the appropriate manual page for details (**man 8 kea-admin**).

4.3 Supported Backends

The following table presents the capabilities of available backends. Please refer to the specific sections dedicated to each backend to better understand their capabilities and limitations. Choosing the right backend may be essential for the success of your deployment.

Feature	Memfile	MySQL	PostgreSQL	CQL (Cassandra)
Status	Stable	Stable	Stable	Experimental
Data format	CSV file	SQL RMDB	SQL RMDB	NoSQL database (Cassandra)
Leases	yes	yes	yes	yes
Host Reservations	no	yes	yes	yes
Options defined on per host basis	no	yes	yes	yes
Configuration Backend	no	yes	no	no

Table 4.1: List of available backends

4.3.1 memfile

The memfile backend is able to store lease information, but is not able to store host reservation details; these must be stored in the configuration file. (There are no plans to add a host reservations storage capability to this backend.)

No special initialization steps are necessary for the memfile backend. During the first run, both **kea-dhcp4** and **kea-dhcp6** will create an empty lease file if one is not present. Necessary disk-write permission is required.

4.3.1.1 Upgrading Memfile Lease Files from an Earlier Version of Kea

There are no special steps required to upgrade memfile lease files from an earlier version of Kea to a new version of Kea. During startup the servers will check the schema version of the lease files against their own. If there is a mismatch, the servers will automatically launch the LFC process to convert the files to the server's schema version. While this mechanism is primarily meant to ease the process of upgrading to newer versions of Kea, it can also be used for downgrading should the need arise. When upgrading, any values not present in the original lease files will be assigned appropriate default values. When downgrading, any data present in the files but not in the server's schema will be dropped. If you wish to convert the files manually prior to starting the servers, you may do so by running the LFC process yourself. See Chapter 13 for more information.



4.3.2 MySQL

MySQL is able to store leases, host reservations, options defined on a per-host basis and a subset of the server configuration parameters (serving as a configuration backend). This section can be safely ignored if you choose to store the data in other backends.

4.3.2.1 First-Time Creation of the MySQL Database

If you are setting the MySQL database for the first time, you need to create the database area within MySQL and set up the MySQL user ID under which Kea will access the database. This needs to be done manually; **kea-admin** is not able to do this for you.

To create the database:

1. Log into MySQL as "root":

```
$ mysql -u root -p
Enter password:
mysql>
```

2. Create the MySQL database:

```
mysql> CREATE DATABASE database-name;
```

(*database-name* is the name you have chosen for the database.)

3. Create the user under which Kea will access the database (and give it a password), then grant it access to the database tables:

```
mysql> CREATE USER 'user-name'@'localhost' IDENTIFIED BY 'password';
mysql> GRANT ALL ON database-name.* TO 'user-name'@'localhost';
```

(*user-name* and *password* are the user ID and password you are using to allow Kea's access to the MySQL instance. All apostrophes in the command lines above are required.)

4. At this point, you may elect to create the database tables. (Alternatively, you can exit MySQL and create the tables using the **kea-admin** tool, as explained below.) To do this:

```
mysql> CONNECT database-name;
mysql> SOURCE path-to-kea/share/kea/scripts/mysql/dhcpdb_create.mysql
```

(*path-to-kea* is the location where you installed Kea.)

5. Exit MySQL:

```
mysql> quit
Bye
$
```

If you elected not to create the tables in Step 4, you can do so now by running the **kea-admin** tool:

```
$ kea-admin db-init mysql -u database-user -p database-password -n database-name
```

(Do not do this if you did create the tables in Step 4.) **kea-admin** implements rudimentary checks; it will refuse to initialize a database that contains any existing tables. If you want to start from scratch, you must remove all data manually. (This process is a manual operation on purpose, to avoid possibly irretrievable mistakes by **kea-admin**.)



4.3.2.2 Upgrading a MySQL Database from an Earlier Version of Kea

Sometimes a new Kea version may use a newer database schema, so the existing database will need to be upgraded. This can be done using the **kea-admin db-upgrade** command.

To check the current version of the database, use the following command:

```
$ kea-admin db-version mysql -u database-user -p database-password -n database-name
```

(See Section 4.1 for a discussion about versioning.) If the version does not match the minimum required for the new version of Kea (as described in the release notes), the database needs to be upgraded.

Before upgrading, please make sure that the database is backed up. The upgrade process does not discard any data, but depending on the nature of the changes, it may be impossible to subsequently downgrade to an earlier version. To perform an upgrade, issue the following command:

```
$ kea-admin db-upgrade mysql -u database-user -p database-password -n database-name
```

4.3.3 PostgreSQL

PostgreSQL is able to store leases, host reservations, and options defined on a per-host basis. This step can be safely ignored if you are using other database backends.

4.3.3.1 First-Time Creation of the PostgreSQL Database

The first task is to create both the database and the user under which the servers will access it. A number of steps are required:

1. Log into PostgreSQL as "root":

```
$ sudo -u postgres psql postgres
Enter password:
postgres=#
```

2. Create the database:

```
postgres=# CREATE DATABASE database-name;
CREATE DATABASE
postgres=#
```

(*database-name* is the name you have chosen for the database.)

3. Create the user under which Kea will access the database (and give it a password), then grant it access to the database:

```
postgres=# CREATE USER user-name WITH PASSWORD 'password' ;
CREATE ROLE
postgres=# GRANT ALL PRIVILEGES ON DATABASE database-name TO user-name;
GRANT
postgres=#
```

4. Exit PostgreSQL:

```
postgres=# \q
Bye
$
```

5. At this point you are ready to create the database tables. This can be done using the **kea-admin** tool as explained in the next section (recommended), or manually. To create the tables manually, enter the following command. Note that PostgreSQL will prompt you to enter the new user's password you specified in Step 3. When the command completes, you will be returned to the shell prompt. You should see output similar to the following:



```
$ psql -d database-name -U user-name -f path-to-kea/share/kea/scripts/pgsql/ ↵
dhcpdb_create.pgsql
Password for user user-name:
CREATE TABLE
CREATE INDEX
CREATE INDEX
CREATE TABLE
CREATE INDEX
CREATE TABLE
START TRANSACTION
INSERT 0 1
INSERT 0 1
INSERT 0 1
COMMIT
CREATE TABLE
START TRANSACTION
INSERT 0 1
COMMIT
$
```

(*path-to-kea* is the location where you installed Kea.)

If instead you encounter an error like:

```
psql: FATAL: no pg_hba.conf entry for host "[local]", user "user-name", database " ↵
database-name", SSL off
```

... you will need to alter the PostgreSQL configuration. Kea uses password authentication when connecting to the database and must have the appropriate entries added to PostgreSQL's `pg_hba.conf` file. This file is normally located in the primary data directory for your PostgreSQL server. The precise path may vary depending on your operating system and version, but the default location for PostgreSQL 9.3 on Centos 6.5 is: `/var/lib/pgsql/9.3/data/pg_hba.conf`.

Assuming Kea is running on the same host as PostgreSQL, adding lines similar to the following should be sufficient to provide password-authenticated access to Kea's database:

local	<i>database-name</i>	<i>user-name</i>		password
host	<i>database-name</i>	<i>user-name</i>	127.0.0.1/32	password
host	<i>database-name</i>	<i>user-name</i>	:::1/128	password

These edits are primarily intended as a starting point, and are not a definitive reference on PostgreSQL administration or database security. Please consult your PostgreSQL user manual before making these changes, as they may expose other databases that you run. It may be necessary to restart PostgreSQL in order for the changes to take effect.

4.3.3.2 Initialize the PostgreSQL Database Using `kea-admin`

If you elected not to create the tables manually, you can do so now by running the **kea-admin** tool:

```
$ kea-admin db-init pgsql -u database-user -p database-password -n database-name
```

Do not do this if you already created the tables manually. **kea-admin** implements rudimentary checks; it will refuse to initialize a database that contains any existing tables. If you want to start from scratch, you must remove all data manually. (This process is a manual operation on purpose, to avoid possibly irretrievable mistakes by **kea-admin**.)

4.3.3.3 Upgrading a PostgreSQL Database from an Earlier Version of Kea

The PostgreSQL database schema can be upgraded using the same tool and commands as described in Section 4.3.2.2, with the exception that the "pgsql" database backend type must be used in the commands.

Use the following command to check the current schema version:



```
$ kea-admin db-version pgsql -u database-user -p database-password -n database-name
```

Use the following command to perform an upgrade:

```
$ kea-admin db-upgrade pgsql -u database-user -p database-password -n database-name
```

4.3.4 Cassandra

Cassandra (sometimes for historical reasons referred to in documentation and commands as CQL) is the newest backend added to Kea; initial development was contributed by Deutsche Telekom. The Cassandra backend is able to store leases, host reservations, and options defined on a per-host basis.

Cassandra must be properly set up if you want Kea to store information in it. This section can be safely ignored if you choose to store the data in other backends.

4.3.4.1 First-Time Creation of the Cassandra Database

If you are setting up the Cassandra database for the first time, you need to create the keyspace area within it. This needs to be done manually; **kea-admin** cannot do this for you.

To create the database:

1. Export CQLSH_HOST environment variable:

```
$ export CQLSH_HOST=localhost
```

2. Log into CQL:

```
$ cqlsh
cql>
```

3. Create the CQL keyspace:

```
cql> CREATE KEYSPACE keyspace-name WITH replication = {'class' : 'SimpleStrategy', '↵
  replication_factor' : 1};
```

(*keyspace-name* is the name you have chosen for the keyspace)

4. At this point, you may elect to create the database tables. (Alternatively, you can exit Cassandra and create the tables using the **kea-admin** tool, as explained below.) To do this:

```
cqslh -k keyspace-name -f path-to-kea/share/kea/scripts/cql/dhcpdb_create.cql
```

(*path-to-kea* is the location where you installed Kea)

If you elected not to create the tables in Step 4, you can do so now by running the **kea-admin** tool:

```
$ kea-admin db-init cql -n database-name
```

(Do not do this if you did create the tables in Step 4.) **kea-admin** implements rudimentary checks; it will refuse to initialize a database that contains any existing tables. If you want to start from scratch, you must remove all data manually. (This process is a manual operation on purpose, to avoid possibly irretrievable mistakes by **kea-admin**.)



4.3.4.2 Upgrading a Cassandra Database from an Earlier Version of Kea

Sometimes a new Kea version may use a newer database schema, so the existing database will need to be upgraded. This can be done using the **kea-admin db-upgrade** command.

To check the current version of the database, use the following command:

```
$ kea-admin db-version cql -n database-name
```

(See Section 4.1 for a discussion about versioning.) If the version does not match the minimum required for the new version of Kea (as described in the release notes), the database needs to be upgraded.

Before upgrading, please make sure that the database is backed up. The upgrade process does not discard any data, but depending on the nature of the changes, it may be impossible to subsequently downgrade to an earlier version. To perform an upgrade, issue the following command:

```
$ kea-admin db-upgrade cql -n database-name
```

4.3.5 Using Read-Only Databases with Host Reservations

If a read-only database is used for storing host reservations, Kea must be explicitly configured to operate on the database in read-only mode. Sections Section 8.2.3.2 and Section 9.2.3.2 describe when such configuration may be required and how to configure Kea to operate in this way.

4.3.6 Limitations Related to the Use of SQL Databases

4.3.6.1 Year 2038 Issue

The lease expiration time is stored in the SQL database for each lease as a timestamp value. Kea developers observed that the MySQL database doesn't accept timestamps beyond 2147483647 seconds (maximum signed 32-bit number) from the beginning of the Unix epoch (00:00:00 on 1 January 1970). Some versions of PostgreSQL do accept greater values, but the value is altered when it is read back. For this reason, the lease database backends put a restriction on the maximum timestamp to be stored in the database, which is equal to the maximum signed 32-bit number. This effectively means that the current Kea version cannot store leases whose expiration time is later than 2147483647 seconds since the beginning of the epoch (around year 2038). This will be fixed when the database support for longer timestamps is available.



Chapter 5

Kea Configuration

Kea uses JSON structures to represent server configurations. The following sections describe how the configuration structures are organized.

5.1 JSON Configuration

JSON is the notation used throughout the Kea project. The most obvious usage is for the configuration file, but JSON is also used for sending commands over the Management API (see Chapter 17) and for communicating between DHCP servers and the DDNS update daemon.

Typical usage assumes that the servers are started from the command line, either directly or using a script, e.g. `keactrl`. The configuration file is specified upon startup using the `-c` parameter.

5.1.1 JSON Syntax

Configuration files for the DHCPv4, DHCPv6, DDNS, Control Agent, and Netconf modules are defined in an extended JSON format. Basic JSON is defined in [RFC 7159](#) and [ECMA 404](#). In particular, the only boolean values allowed are `true` or `false` (all lowercase). The capitalized versions (`True` or `False`) are not accepted.

Kea components use an extended JSON with additional features allowed:

- shell comments: any text after the hash (`#`) character is ignored.
- C comments: any text after the double slashes (`//`) character is ignored.
- Multiline comments: any text between `/*` and `*/` is ignored. This commenting can span multiple lines.
- File inclusion: JSON files can include other JSON files by using a statement of the form `<?include "file.json"?>`.

The configuration file consists of a single object (often colloquially called a map) started with a curly bracket. It comprises one or more of the `"Dhcp4"`, `"Dhcp6"`, `"DhcpDdns"`, `"Control-agent"` and `"Netconf"` objects. It is possible to define additional elements but they will be ignored.

A very simple configuration for DHCPv4 could look like this:

```
# The whole configuration starts here.
{
  # DHCPv4 specific configuration starts here.
  "Dhcp4": {
    "interfaces-config": {
      "interfaces": [ "eth0" ],
      "dhcp-socket-type": "raw"
    }
  }
}
```



```
    },
    "valid-lifetime": 4000,
    "renew-timer": 1000,
    "rebind-timer": 2000,
    "subnet4": [{
        "pools": [ { "pool": "192.0.2.1-192.0.2.200" } ],
        "subnet": "192.0.2.0/24"
    }],

    # Now loggers are inside the DHCPv4 object.
    "loggers": [{
        "name": "*",
        "severity": "DEBUG"
    }]
}

# The whole configuration structure ends here.
}
```

More examples are available in the installed `share/doc/kea/examples` directory.

Note

The "Logging" element is removed in Kea 1.6.0 and its contents (the "loggers" object) moved inside the configuration objects (maps) for respective Kea modules. For example: the "Dhcp4" map contains the "loggers" object specifying logging configuration for the DHCPv4 server. Backward compatibility is maintained until at least Kea 1.7.0 release: it will be possible to specify "Logging" object at the top configuration level and "loggers" objects at the module configuration level. Ultimately, support for the top-level "Logging" object will be removed.

The specification of several supported elements (e.g. "Dhcp4", "Dhcp6") in a single configuration file can be confusing and works badly with the commands that fetch and write new configurations. Support for it will be removed in a future release of Kea, after which each component will require its own configuration file.

To avoid repetition of mostly similar structures, examples in the rest of this guide will showcase only the subset of parameters appropriate for a given context. For example, when discussing the IPv6 subnets configuration in DHCPv6, only subnet6 parameters will be mentioned. It is implied that the remaining elements (the global map that holds Dhcp6 and Logging) are present, but they are omitted for clarity. Usually, locations where extra parameters may appear are denoted by an ellipsis (...).

5.1.2 Simplified Notation

It is sometimes convenient to refer to a specific element in the configuration hierarchy. Each hierarchy level is separated by a slash. If there is an array, a specific instance within that array is referenced by a number in square brackets (with numbering starting at zero). For example, in the above configuration the valid-lifetime in the Dhcp4 component can be referred to as `Dhcp4/valid-lifetime` and the pool in the first subnet defined in the DHCPv4 configuration as `Dhcp4/subnet4[0]/pool`.

5.2 Kea Configuration Backend

5.2.1 Applicability

Kea Configuration Backend (abbreviated as CB) is a feature first introduced in 1.6.0 release, which provides Kea servers with the ability to manage and fetch their configuration from one or more databases. In the documentation, the term "Configuration Backend" may also refer to the particular Kea module providing support to manage and fetch the configuration information from the particular database type. For example: MySQL Configuration Backend is the logic implemented within the "mysql_cb" hooks library which provides a complete set of functions to manage and fetch the configuration information from the MySQL database.



In small deployments, e.g. those comprising a single DHCP server instance with limited and infrequently changing number of subnets, it may be impractical to use the CB as a configuration repository because it requires additional third party software to be installed and configured - in particular the MySQL server and MySQL client. Once the number of DHCP servers and/or the number of managed subnets in the network grows, the usefulness of the CB becomes obvious.

A good example is a pair of the Kea DHCP servers which can be configured to support High Availability as described in Section 15.4.9. The configurations of both servers are almost exactly the same. They may differ by the server identifier and designation of the server as a primary or standby (or secondary). They may also differ by the interfaces configuration. Typically, the subnets, shared networks, option definitions, global parameters are the same for both servers and can be sourced from a single database instance to both Kea servers.

Using the database as a single source of configuration for subnets and/or other configuration information supported by the CB has the advantage that any modifications to the configuration in the database is automatically applied to both servers.

Another case when the centralized configuration repository is desired is in deployments including large number of the DHCP servers, possibly using a common lease database to provide redundancy. The new servers can be added to the pool frequently to fulfil growing scalability requirements. Adding the new server does not require replicating the entire configuration to the new server when common database is used.

Using the database as a configuration repository for Kea servers also brings other benefits, such as:

- an ability to use database specific tools to access the configuration information,
- an ability to create customized statistics based on the information stored in the database,
- an ability to backup the configuration information using the database builtin replication mechanisms.

5.2.2 CB Capabilities and Limitations

Kea CB has been introduced in the 1.6.0 release, but this implementation comes with a number of limitations being the result of the overall complexity of this feature and the development time constraints. This feature will evolve over time and the new capabilities will be added in subsequent releases. In this section we present the limitations of the CB, present in the current Kea 1.6.0 release:

- Kea CB is currently supported for the MySQL database only.
- Kea CB is only supported for DHCPv4 and DHCPv6 servers. Neither Control Agent nor D2 daemon can be configured via the database.
- Configuration to be stored for the DHCP servers includes: global parameters, option definitions, global options, shared networks and subnets. Other configuration parameters are not stored in the database at the moment. They have to be configured via the JSON configuration file.

Note

We strongly recommend to not duplicate the configuration information in the file and the database. For example, when specifying subnets for the DHCP server, please store them in the configuration backend or in the configuration file, not in both places. Storing some subnets in the database and other in the file may put you at risk of potential configuration conflicts. Note that the configuration from the database takes precedence over the configuration from the file, thus it is possible that parts of the configuration specified in the file may be overridden.

Note

It is recommended that **subnet_cmds** hooks library is not used to manage the subnets when the configuration backend is used as a source of information about the subnets. The **subnet_cmds** hooks library modifies the local subnets configuration (in the server's memory), not in the database. Use the **cb_cmds** hooks library to manage the subnets information in the database instead.



5.2.3 CB Components

In order to use the Kea CB feature, the Kea 1.6.0 version or later is required. The **mysql_cb** open source hooks library implementing the Configuration Backend for MySQL must be compiled and loaded by the DHCP servers. This hooks library is compiled when the `--with-mysql` configuration switch is used during Kea build. The MySQL C client libraries must be installed as explained in the Section 3.5.

Note

Any existing MySQL schema must be upgraded to the latest schema required by the particular Kea version using **kea-admin** tool described in Section 4.2.

The **cb_cmds** premium hooks library is available to ISC paid supported customers, which provides a complete set of commands to manage the servers' configuration information within the database. This library can be attached to both DHCPv4 and DHCPv6 server instances. It is still possible to manage the configuration information without the **cb_cmds** hooks library with commonly available tools such as MySQL Workbench or command line MySQL client, by directly working with the database.

Refer to the Section 15.4.8 for the details regarding the **cb_cmds** hooks library.

The DHCPv4 and DHCPv6 server specific configuration of the CB as well as the list of supported configuration parameters can be found in the Section 8.14 and Section 9.19 respectively.

5.2.4 Configuration Sharing and Server Tags

The configuration database is designed to store the configuration information for multiple Kea servers. Depending on the use case, the entire configuration may be shared by all servers, parts of the configuration may be shared by multiple servers and the rest of the configuration may be different for these servers or, finally, each server may have its own non-shared configuration.

The configuration elements in the database are associated with the servers by "server tags". The server tag is an arbitrary string holding the name of the Kea server instance. The tags of the DHCPv4 and DHCPv6 servers are independent in the database, i.e. the same server tag can be created for the DHCPv4 and the DHCPv6 server respectively.

The server definition, which consists of the server tag and the server description, must be stored in the configuration database prior to creating the dedicated configuration for that server. In cases when all servers use the same configuration, e.g. a pair of servers running as the High Availability peers, there is no need to configure the server tags for these servers in the database. The database by default includes the logical server **all**, which is used as a keyword to indicate that the particular piece of configuration must be shared between all servers connecting to the database. The **all** server can't be deleted or modified. It is not even returned among other servers as a result of the **remote-server[46]-get-all** commands. Also, slightly different rules may apply to "all" keyword than to any user defined server when running the commands provided by the **cb_cmds** hooks library (see Section 15.4.8 for details).

Note

In the simplest case there are no server tags defined in the configuration database and all connecting servers will get the same configuration regardless of the server tag they are using. The server tag that the particular Kea instance presents to the database to fetch its configuration is specified in the Kea configuration file, using the **config-control** map (please refer to the Section 8.14.2 and Section 9.19.2 for details).

All Kea instances presenting the same server tag to the configuration database are given the same configuration. It is the administrator's choice whether multiple Kea instances use the same server tag or each Kea instance is using a different sever tag. Also, there is no requirement that the instances running on the same physical or virtual machine use the same server tag. It is even possible to configure the Kea server without assigning it a server tag. In such case the server will be given the configuration specified for "all" servers.

In order to differentiate the configurations between the Kea servers, a collection of the server tags used by the servers must be stored in the database. For the DHCPv4 and DHCPv6 servers, it can be done using the commands described in Section 15.4.8.2.5 and Section 15.4.8.2.5 [239]. Next, the server tags can be used to associate the configuration information with the servers.



However, it is important to note that some DHCP configuration elements may be associated with multiple server tags and other configuration elements may be associated with exactly one server tag. The former configuration elements are referred to as shareable configuration elements and the latter are referred to as non-shareable configuration elements. The Section 8.14 and Section 9.19 list the DHCP specific shareable and non-shareable configuration elements. However, in this section we want to briefly explain the difference between them.

The shareable configuration element is the one having some unique property identifying it and which instance may appear only once in the database. An example of the shareable DHCP element is a subnet instance. The subnet is a part of the network topology and we assume that the particular subnet may have only one definition within this network. The subnet has two unique identifiers: subnet id and the subnet prefix. The subnet identifier is used in Kea to uniquely identify the subnet and to connect it with other configuration elements, e.g. in host reservations. The subnet identifier uniquely identifies the subnet within the network. Some commands provided by the `cb_cmds` hooks library allow for accessing the subnet information by subnet identifier (or prefix) and explicitly prohibit using the server tag to access the subnet. This is because, in a general case, the subnet definition is associated with multiple servers rather than single server. In fact, it may even be associated with no servers (unassigned). Still, the unassigned subnet has an identifier and prefix which can be used to access the subnet.

A shareable configuration element may be associated with multiple servers, one server or no servers. Deletion of the server which is associated with the shareable element does not cause the deletion of the shareable element. It merely deletes the association of the deleted server with the element.

Unlike the shareable element, the non-shareable element must not be explicitly associated with more than one server and must not exist after the server is deleted (must not remain unassigned). The non-shareable element only exists within the context of the server. An example of the non-shareable element in DHCP is a global parameter, e.g. **renew-timer**. The renew timer is the value to be used by the particular server and only this server. Other servers may have their respective renew timers set to the same or different value. The renew timer is the parameter which has no unique identifier by which it could be accessed, modified or otherwise used. The global parameters like the renew timer can be accessed by the parameter name and the tag of the server for which they are configured. For example: the commands described in Section 15.4.8.2.7 allow for fetching the value of the global parameter by the parameter name and the server name. Getting the global parameter only by its name (without specifying the server tag) is not possible because there may be many global parameters with the given name in the database.

When the server associated with a non-shareable configuration element is deleted, the configuration element is automatically deleted from the database along with the server because the non-shareable element must be always assigned to some server (or the logical server "all").

The terms "shareable" and "non-shareable" only apply to the associations with user defined servers. All configuration elements associated with the logical server "all" are by definition shareable. For example: the **renew-timer** associated with "all" servers is used by all servers connecting to the database which don't have their specific renew timers defined. In the special case, when none of the configuration elements are associated with user defined servers, the entire configuration in the database is shareable because all its pieces belong to "all" servers.

Note

Be very careful when associating the configuration elements with different server tags. The configuration backend doesn't protect you against some possible misconfigurations that may arise from the wrong server tags' assignments. For example: if you assign a shared network to one server and the subnets belonging to this shared network to another server, the servers will fail upon trying to fetch and use this configuration. The server fetching the subnets will be aware that the subnets are associated with the shared network but the shared network will not be found by this server as it doesn't belong to it. In such case, both the shared network and the subnets should be assigned to the same set of servers.



Chapter 6

Managing Kea with keactrl

6.1 Overview

keactrl is a shell script which controls the startup, shutdown, and reconfiguration of the Kea servers (**kea-dhcp4**, **kea-dhcp6**, **kea-dhcp-ddns**, **kea-ctrl-agent**, and **kea-netconf**). It also provides the means for checking the current status of the servers and determining the configuration files in use.

6.2 Command Line Options

keactrl is run as follows:

```
keactrl <command> [-c keactrl-config-file] [-s server[,server,...]]
```

<command> is the one of the commands described in Section 6.4.

The optional **-c keactrl-config-file** switch allows specification of an alternate **keactrl** configuration file. (**--ctrl-config** is a synonym for **-c**.) In the absence of **-c**, **keactrl** will use the default configuration file `[kea-install-dir]/etc/kea/keactrl.conf`.

The optional **-s server[,server,...]** switch selects the servers to which the command is issued. (**--server** is a synonym for **-s**.) If absent, the command is sent to all servers enabled in the keactrl configuration file. If multiple servers are specified, they should be separated by commas with no intervening spaces.

6.3 The keactrl Configuration File

Depending on requirements, not all of the available servers need to be run. The keactrl configuration file sets which servers are enabled and which are disabled. The default configuration file is `[kea-install-dir]/etc/kea/keactrl.conf`, but this can be overridden on a per-command basis using the **-c** switch.

The contents of `keactrl.conf` are:

```
# This is a configuration file for keactrl script which controls
# the startup, shutdown, reconfiguration and gathering the status
# of the Kea's processes.

# prefix holds the location where the Kea is installed.
prefix=@prefix@

# Location of Kea configuration file.
kea_dhcp4_config_file=@sysconfdir@/@PACKAGE@/kea-dhcp4.conf
```



```
kea_dhcp6_config_file=@sysconfdir@/@PACKAGE@/kea-dhcp6.conf
kea_dhcp_ddns_config_file=@sysconfdir@/@PACKAGE@/kea-dhcp-ddns.conf
kea_ctrl_agent_config_file=@sysconfdir@/@PACKAGE@/kea-ctrl-agent.conf
kea_netconf_config_file=@sysconfdir@/@PACKAGE@/kea-netconf.conf

# Location of Kea binaries.
exec_prefix=@exec_prefix@
dhcp4_srv=@sbindir@/kea-dhcp4
dhcp6_srv=@sbindir@/kea-dhcp6
dhcp_ddns_srv=@sbindir@/kea-dhcp-ddns
ctrl_agent_srv=@sbindir@/kea-ctrl-agent
netconf_srv=@sbindir@/kea-netconf

# Start DHCPv4 server?
dhcp4=yes

# Start DHCPv6 server?
dhcp6=yes

# Start DHCP DDNS server?
dhcp_ddns=no

# Start Control Agent?
ctrl_agent=yes

# Start Netconf?
netconf=no

# Be verbose?
kea_verbose=no
```

Note

In the example above, strings of the form @something@ are replaced by the appropriate values when Kea is installed.

The *dhcp4*, *dhcp6*, *dhcp_ddns*, *ctrl_agent*, and *netconf* parameters set to "yes" will configure **keactrl** to manage (start, reconfigure) all servers, i.e. **kea-dhcp4**, **kea-dhcp6**, **kea-dhcp-ddns**, **kea-ctrl-agent**, and **kea-netconf**. When any of these parameters is set to "no," the **keactrl** will ignore the corresponding server when starting or reconfiguring Kea. Some daemons (ddns and netconf) are disabled by default.

By default, Kea servers managed by **keactrl** are located in `[kea-install-dir]/sbin`. This should work for most installations. If the default location needs to be altered for any reason, the paths specified with the *dhcp4_srv*, *dhcp6_srv*, *dhcp_ddns_srv*, *ctrl_agent_srv*, and *netconf_srv* parameters should be modified.

The *kea_verbose* parameter specifies the verbosity of the servers being started. When *kea_verbose* is set to "yes" the logging level of the server is set to DEBUG. Modification of the logging severity in a configuration file, as described in Chapter 18, will have no effect as long as the *kea_verbose* is set to "yes." Setting it to "no" will cause the server to use the logging levels specified in the Kea configuration file. If no logging configuration is specified, the default settings will be used.

Note

The verbosity for the server is set when it is started. Once started, the verbosity can be only changed by stopping the server and starting it again with the new value of the *kea_verbose* parameter.

6.4 Commands

The following commands are supported by **keactrl**:



- **start** - starts selected servers.
- **stop** - stops all running servers.
- **reload** - triggers reconfiguration of the selected servers by sending the SIGHUP signal to them.
- **status** - returns the status of the servers (active or inactive) and the names of the configuration files in use.
- **version** - prints out the version of the keactrl tool itself, together with the versions of the Kea daemons.

Typical output from **keactrl** when starting the servers looks similar to the following:

```
$ keactrl start
INFO/keactrl: Starting kea-dhcp4 -c /usr/local/etc/kea/kea-dhcp4.conf -d
INFO/keactrl: Starting kea-dhcp6 -c /usr/local/etc/kea/kea-dhcp6.conf -d
INFO/keactrl: Starting kea-dhcp-ddns -c /usr/local/etc/kea/kea-dhcp-ddns.conf -d
INFO/keactrl: Starting kea-ctrl-agent -c /usr/local/etc/kea/kea-ctrl-agent.conf -d
INFO/keactrl: Starting kea-netconf -c /usr/local/etc/kea/kea-netconf.conf -d
```

Kea's servers create PID files upon startup. These files are used by keactrl to determine whether a given server is running. If one or more servers are running when the start command is issued, the output will look similar to the following:

```
$ keactrl start
INFO/keactrl: kea-dhcp4 appears to be running, see: PID 10918, PID file: /usr/local/var/run ←
  /kea/kea.kea-dhcp4.pid.
INFO/keactrl: kea-dhcp6 appears to be running, see: PID 10924, PID file: /usr/local/var/run ←
  /kea/kea.kea-dhcp6.pid.
INFO/keactrl: kea-dhcp-ddns appears to be running, see: PID 10930, PID file: /usr/local/var ←
  /run/kea/kea.kea-dhcp-ddns.pid.
INFO/keactrl: kea-ctrl-agent appears to be running, see: PID 10931, PID file: /usr/local/ ←
  var/run/kea/kea.kea-ctrl-agent.pid.
INFO/keactrl: kea-netconf appears to be running, see: PID 10123, PID file: /usr/local/var/ ←
  run/kea/kea.kea-netconf.pid.
```

During normal shutdowns these PID files are deleted. They may, however, be left over as remnants following a system crash. It is possible, though highly unlikely, that upon system restart the PIDs they contain may actually refer to processes unrelated to Kea. This condition will cause keactrl to decide that the servers are running, when in fact they are not. In such a case the PID files as listed in the keactrl output must be manually deleted.

The following command stops all servers:

```
$ keactrl stop
INFO/keactrl: Stopping kea-dhcp4...
INFO/keactrl: Stopping kea-dhcp6...
INFO/keactrl: Stopping kea-dhcp-ddns...
INFO/keactrl: Stopping kea-ctrl-agent...
INFO/keactrl: Stopping kea-netconf...
```

Note that the **stop** command will attempt to stop all servers regardless of whether they are "enabled" in the `keactrl.conf`. If any of the servers are not running, an informational message is displayed as in the **stop** command output below.

```
$ keactrl stop
INFO/keactrl: kea-dhcp4 isn't running.
INFO/keactrl: kea-dhcp6 isn't running.
INFO/keactrl: kea-dhcp-ddns isn't running.
INFO/keactrl: kea-ctrl-agent isn't running.
INFO/keactrl: kea-netconf isn't running.
```

As already mentioned, the reconfiguration of each Kea server is triggered by the SIGHUP signal. The **reload** command sends the SIGHUP signal to the servers that are enabled in the **keactrl** configuration file and are currently running. When a server receives the SIGHUP signal it re-reads its configuration file and, if the new configuration is valid, uses the new configuration. A reload is executed as follows:

**\$ keactrl reload**

```
INFO/keactrl: Reloading kea-dhcp4...
INFO/keactrl: Reloading kea-dhcp6...
INFO/keactrl: Reloading kea-dhcp-ddns...
INFO/keactrl: Reloading kea-ctrl-agent...
```

If any of the servers are not running, an informational message is displayed as in the **reload** command output below. Note that as of version 1.5.0, kea-netconf does not support the SIGHUP signal. If its configuration has changed, please stop and restart it for the change to take effect. This limitation will be removed in a future release.

\$ keactrl stop

```
INFO/keactrl: kea-dhcp4 isn't running.
INFO/keactrl: kea-dhcp6 isn't running.
INFO/keactrl: kea-dhcp-ddns isn't running.
INFO/keactrl: kea-ctrl-agent isn't running.
INFO/keactrl: kea-netconf isn't running.
```

Note NETCONF is an optional feature that is disabled by default and can be enabled during compilation. If Kea was compiled without NETCONF support, keactrl will do its best to not bother the user with information about it. The netconf entries will still be present in the keactrl.conf file, but netconf status will not be shown and other commands will ignore it.

Note

Currently **keactrl** does not report configuration failures when the server is started or reconfigured. To check if the server's configuration succeeded, the Kea log must be examined for errors. By default, this is written to the syslog file.

Sometimes it is useful to check which servers are running. The **status** reports this, with typical output that looks like:

\$ keactrl status

```
DHCPv4 server: active
DHCPv6 server: inactive
DHCP DDNS: active
Control Agent: active
Netconf agent: inactive
Kea configuration file: /usr/local/etc/kea/kea.conf
Kea DHCPv4 configuration file: /usr/local/etc/kea/kea-dhcp4.conf
Kea DHCPv6 configuration file: /usr/local/etc/kea/kea-dhcp6.conf
Kea DHCP DDNS configuration file: /usr/local/etc/kea/kea-dhcp-ddns.conf
Kea Control Agent configuration file: /usr/local/etc/kea/kea-ctrl-agent.conf
Kea Netconf configuration file: /usr/local/etc/kea/kea-netconf.conf
keactrl configuration file: /usr/local/etc/kea/keactrl.conf
```

6.5 Overriding the Server Selection

The optional **-s** switch allows the selection of the servers to which the **keactrl** command is issued. For example, the following instructs **keactrl** to stop the **kea-dhcp4** and **kea-dhcp6** servers and leave the **kea-dhcp-ddns** and **kea-ctrl-agent** running:

```
$ keactrl stop -s dhcp4,dhcp6
```

Similarly, the following will start only the **kea-dhcp4** and **kea-dhcp-ddns** servers, but not **kea-dhcp6** or **kea-ctrl-agent**.

```
$ keactrl start -s dhcp4,dhcp_ddns
```



Note that the behavior of the **-s** switch with the **start** and **reload** commands is different to its behavior with the **stop** command. On **start** and **reload**, **keactrl** will check if the servers given as parameters to the **-s** switch are enabled in the **keactrl** configuration file; if not, the server will be ignored. For **stop**, however, this check is not made; the command is applied to all listed servers, regardless of whether they have been enabled in the file.

The following keywords can be used with the **-s** command line option:

- **dhcp4** for **kea-dhcp4**.
- **dhcp6** for **kea-dhcp6**.
- **dhcp_ddns** for **kea-dhcp-ddns**.
- **ctrl_agent** for **kea-ctrl-agent**.
- **netconf** for **kea-netconf**.
- **all** for all servers (default).



Chapter 7

The Kea Control Agent

7.1 Overview

The Kea Control Agent (CA) is a daemon which exposes a RESTful control interface for managing Kea servers. The daemon can receive control commands over HTTP and either forward these commands to the respective Kea servers or handle these commands on its own. The determination whether the command should be handled by the CA or forwarded is made by checking the value of the "service" parameter, which may be included in the command from the controlling client. The details of the supported commands, as well as their structures, are provided in [Chapter 17](#).

The CA can use hook libraries to provide support for additional commands or custom behavior of existing commands. Such hook libraries must implement callouts for the "control_command_receive" hook point. Details about creating new hook libraries and supported hook points can be found in the [Kea Developer's Guide](#).

The CA processes received commands according to the following algorithm:

- Pass command into any installed hooks (regardless of service value(s)). If the command is handled by a hook, return the response.
- If the service specifies one more or services, forward the command to the specified services and return the accumulated responses.
- If the service is not specified or is an empty list, handle the command if the CA supports it.

7.2 Configuration

The following example demonstrates the basic CA configuration.

```
{
  "Control-agent": {
    "http-host": "10.20.30.40",
    "http-port": 8080,

    "control-sockets": {
      "dhcp4": {
        "comment": "main server",
        "socket-type": "unix",
        "socket-name": "/path/to/the/unix/socket-v4"
      },
      "dhcp6": {
        "socket-type": "unix",
        "socket-name": "/path/to/the/unix/socket-v6",
        "user-context": { "version": 3 }
      }
    }
  }
}
```



```
    },
    "d2": {
      "socket-type": "unix",
      "socket-name": "/path/to/the/unix/socket-d2"
    },
  },
  "hooks-libraries": [
    {
      "library": "/opt/local/control-agent-commands.so",
      "parameters": {
        "param1": "foo"
      }
    }
  ],
  "loggers": [ {
    "name": "kea-ctrl-agent",
    "severity": "INFO"
  } ]
}
```

The **http-host** and **http-port** parameters specify an IP address and port to which HTTP service will be bound. In the example configuration provided above, the RESTful service will be available under the URL of **http://10.20.30.40:8080/**. If these parameters are not specified, the default URL is **http://127.0.0.1:8000/**

As mentioned in Section 7.1, the CA can forward received commands to the Kea servers for processing. For example, **config-get** is sent to retrieve the configuration of one of the Kea services. When the CA receives this command, including a **service** parameter indicating that the client desires to retrieve the configuration of the DHCPv4 server, the CA forwards this command to that server and passes the received response back to the client. More about the **service** parameter and the general structure of commands can be found in Chapter 17.

The CA uses UNIX domain sockets to forward control commands and receive responses from other Kea services. The **dhcp4**, **dhcp6**, and **d2** maps specify the files to which UNIX domain sockets are bound. In the configuration above, the CA will connect to the DHCPv4 server via **/path/to/the/unix/socket-v4** to forward the commands to it. Obviously, the DHCPv4 server must be configured to listen to connections via this same socket. In other words, the command socket configuration for the DHCPv4 server and the CA (for this server) must match. Consult Section 8.9, Section 9.14 and Section 12.3.2 to learn how the socket configuration is specified for the DHCPv4, DHCPv6 and D2 services.



Warning

"dhcp4-server", "dhcp6-server" and "d2-server" were renamed to "dhcp4", "dhcp6" and "d2" respectively in Kea 1.2. If you are migrating from Kea 1.2, you must modify your CA configuration to use this new naming convention.

User contexts can store arbitrary data as long as they are in valid JSON syntax and their top-level element is a map (i.e. the data must be enclosed in curly brackets). Some hook libraries may expect specific formatting; please consult the relevant hook library documentation for details.

User contexts can be specified on either global scope, control socket, or loggers. One other useful feature is the ability to store comments or descriptions; the parser translates a "comment" entry into a user context with the entry, which allows a comment to be attached within the configuration itself.

Hooks libraries can be loaded by the Control Agent in the same way as they are loaded by the DHCPv4 and DHCPv6 servers. The CA currently supports one hook point - 'control_command_receive' - which makes it possible to delegate processing of some commands to the hooks library. The **hooks-libraries** list contains the list of hooks libraries that should be loaded by the CA, along with their configuration information specified with **parameters**.

Please consult Chapter 18 for the details how to configure logging. The CA's root logger's name is **kea-ctrl-agent**, as given in the example above.



7.3 Secure Connections

The Control Agent does not natively support secure HTTP connections like SSL or TLS. In order to setup a secure connection, please use one of the available third-party HTTP servers and configure it to run as a reverse proxy to the Control Agent. Kea has been tested with two major HTTP server implementations working as a reverse proxy: Apache2 and nginx. Example configurations, including extensive comments, are provided in the `doc/examples/https/` directory.

The reverse proxy forwards HTTP requests received over a secure connection to the Control Agent using unsecured HTTP. Typically, the reverse proxy and the Control Agent are running on the same machine, but it is possible to configure them to run on separate machines as well. In this case, security depends on the protection of the communications between the reverse proxy and the Control Agent.

Apart from providing the encryption layer for the control channel, a reverse proxy server is also often used for authentication of the controlling clients. In this case, the client must present a valid certificate when it connects via reverse proxy. The proxy server authenticates the client by checking whether the presented certificate is signed by the certificate authority used by the server.

To illustrate this, the following is a sample configuration for the nginx server running as a reverse proxy to the Kea Control Agent. The server enables authentication of the clients using certificates.

```
# The server certificate and key can be generated as follows:
#
# openssl genrsa -des3 -out kea-proxy.key 4096
# openssl req -new -x509 -days 365 -key kea-proxy.key -out kea-proxy.crt
#
# The CA certificate and key can be generated as follows:
#
# openssl genrsa -des3 -out ca.key 4096
# openssl req -new -x509 -days 365 -key ca.key -out ca.crt
#
# The client certificate needs to be generated and signed:
#
# openssl genrsa -des3 -out kea-client.key 4096
# openssl req -new -key kea-client.key -out kea-client.csr
# openssl x509 -req -days 365 -in kea-client.csr -CA ca.crt \
#     -CAkey ca.key -set_serial 01 -out kea-client.crt
#
# Note that the 'common name' value used when generating the client
# and the server certificates must differ from the value used
# for the CA certificate.
#
# The client certificate must be deployed on the client system.
# In order to test the proxy configuration with 'curl' run
# command similar to the following:
#
# curl -k --key kea-client.key --cert kea-client.crt -X POST \
#     -H Content-Type:application/json -d '{ "command": "list-commands" }' \
#     https://kea.example.org/kea
#
#
# nginx configuration starts here.

events {
}

http {
    # HTTPS server
    server {
        # Use default HTTPS port.
        listen 443 ssl;
        # Set server name.
```



```
server_name kea.example.org;

# Server certificate and key.
ssl_certificate /path/to/kea-proxy.crt;
ssl_certificate_key /path/to/kea-proxy.key;

# Certificate Authority. Client certificate must be signed by the CA.
ssl_client_certificate /path/to/ca.crt;

# Enable verification of the client certificate.
ssl_verify_client on;

# For URLs such as https://kea.example.org/kea, forward the
# requests to http://127.0.0.1:8080.
location /kea {
    proxy_pass http://127.0.0.1:8080;
}
}
```

Note

Note that the configuration snippet provided above is for testing purposes only. It should be modified according to the security policies and best practices of your organization.

When you use an HTTP client without TLS support as **kea-shell**, you can use an HTTP/HTTPS translator such as stunnel in client mode. A sample configuration is provided in the `doc/examples/https/shell/` directory.

7.4 Starting the Control Agent

The CA is started by running its binary and specifying the configuration file it should use. For example:

```
$ ./kea-ctrl-agent -c /usr/local/etc/kea/kea-ctrl-agent.conf
```

It can be started by `keactrl` as well (see Chapter 6).

7.5 Connecting to the Control Agent

For an example of a tool that can take advantage of the RESTful API, see Chapter 19.



Chapter 8

The DHCPv4 Server

8.1 Starting and Stopping the DHCPv4 Server

It is recommended that the Kea DHCPv4 server be started and stopped using **keactrl** (described in Chapter 6); however, it is also possible to run the server directly. It accepts the following command-line switches:

- **-c file** - specifies the configuration file. This is the only mandatory switch.
- **-d** - specifies whether the server logging should be switched to debug/verbose mode. In verbose mode, the logging severity and debuglevel specified in the configuration file are ignored and "debug" severity and the maximum debuglevel (99) are assumed. The flag is convenient for temporarily switching the server into maximum verbosity, e.g. when debugging.
- **-p server-port** - specifies the local UDP port on which the server will listen. This is only useful during testing, as a DHCPv4 server listening on ports other than the standard ones will not be able to handle regular DHCPv4 queries.
- **-P client-port** - specifies the remote UDP port to which the server will send all responses. This is only useful during testing, as a DHCPv4 server sending responses to ports other than the standard ones will not be able to handle regular DHCPv4 queries.
- **-t file** - specifies a configuration file to be tested. Kea-dhcp4 will load it, check it, and exit. During the test, log messages are printed to standard output and error messages to standard error. The result of the test is reported through the exit code (0 = configuration looks ok, 1 = error encountered). The check is not comprehensive; certain checks are possible only when running the server.
- **-v** - displays the Kea version and exits.
- **-V** - displays the Kea extended version with additional parameters and exits. The listing includes the versions of the libraries dynamically linked to Kea.
- **-W** - displays the Kea configuration report and exits. The report is a copy of the `config.report` file produced by `./configure`; it is embedded in the executable binary.

On startup, the server will detect available network interfaces and will attempt to open UDP sockets on all interfaces mentioned in the configuration file. Since the DHCPv4 server opens privileged ports, it requires root access. Make sure you run this daemon as root.

During startup, the server will attempt to create a PID file of the form: `[runstatedir]/kea/[conf name].kea-dhcp4.pid` where:

- **runstatedir**: The value as passed into the build configure script; it defaults to `"/usr/local/var/run"`. Note that this value may be overridden at runtime by setting the environment variable `KEA_PIDFILE_DIR`, although this is intended primarily for testing purposes.
- **conf name**: The configuration file name used to start the server, minus all preceding paths and the file extension. For example, given a pathname of `"/usr/local/etc/kea/myconf.txt"`, the portion used would be `"myconf"`.



If the file already exists and contains the PID of a live process, the server will issue a `DHCP4_ALREADY_RUNNING` log message and exit. It is possible, though unlikely, that the file is a remnant of a system crash and the process to which the PID belongs is unrelated to Kea. In such a case it would be necessary to manually delete the PID file.

The server can be stopped using the `kill` command. When running in a console, the server can also be shut down by pressing `ctrl-c`. It detects the key combination and shuts down gracefully.

8.2 DHCPv4 Server Configuration

8.2.1 Introduction

This section explains how to configure the DHCPv4 server using a configuration file. Before DHCPv4 is started, its configuration file has to be created. The basic configuration is as follows:

```
{
# DHCPv4 configuration starts in this line
"Dhcp4": {

# First we set up global values
  "valid-lifetime": 4000,
  "renew-timer": 1000,
  "rebind-timer": 2000,

# Next we setup the interfaces to be used by the server.
  "interfaces-config": {
    "interfaces": [ "eth0" ]
  },

# And we specify the type of lease database
  "lease-database": {
    "type": "memfile",
    "persist": true,
    "name": "/var/lib/kea/dhcp4.leases"
  },

# Finally, we list the subnets from which we will be leasing addresses.
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [
        {
          "pool": "192.0.2.1 - 192.0.2.200"
        }
      ]
    }
  ]
}
# DHCPv4 configuration ends with the next line
}
```

The following paragraphs provide a brief overview of the parameters in the above example, along with their format. Subsequent sections of this chapter go into much greater detail for these and other parameters.

The lines starting with a hash (`#`) are comments and are ignored by the server; they do not impact its operation in any way.

The configuration starts in the first line with the initial opening curly bracket (or brace). Each configuration must contain an object specifying the configuration of the Kea module using it. In the example above this object is called **Dhcp4**.

**Note**

In the current Kea release it is possible to specify configurations of multiple modules within a single configuration file, but this is not recommended and support for it will be removed in the future releases. The only object, besides the one specifying module configuration, which can (and usually was) included in the same file is **Logging**. However, we don't include this object in the example above for clarity and its content, the list of loggers, should now be inside the **Dhcp4** object instead of the deprecated object.

The Dhcp4 configuration starts with the "**Dhcp4**": { line and ends with the corresponding closing brace (in the above example, the brace after the last comment). Everything defined between those lines is considered to be the Dhcp4 configuration.

In the general case, the order in which those parameters appear does not matter, but there are two caveats. The first one is to remember that the configuration file must be well-formed JSON. That means that the parameters for any given scope must be separated by a comma and there must not be a comma after the last parameter. When reordering a configuration file, keep in mind that moving a parameter to or from the last position in a given scope may also require moving the comma. The second caveat is that it is uncommon — although legal JSON — to repeat the same parameter multiple times. If that happens, the last occurrence of a given parameter in a given scope is used, while all previous instances are ignored. This is unlikely to cause any confusion as there are no real-life reasons to keep multiple copies of the same parameter in your configuration file.

Moving onto the DHCPv4 configuration elements, the first few elements define some global parameters. **valid-lifetime** defines how long the addresses (leases) given out by the server are valid. If nothing changes, a client that got an address is allowed to use it for 4000 seconds. (Note that integer numbers are specified as is, without any quotes around them.) **renew-timer** and **rebind-timer** are values (also in seconds) that define T1 and T2 timers that govern when the client will begin the renewal and rebind procedures.

Note Beginning with Kea 1.6.0 the lease valid lifetime is extended from a single value to a triplet with minimum, default and maximum values using **min-valid-lifetime**, **valid-lifetime** and **max-valid-lifetime**. When the client does not specify a lifetime the default value is used, when it specifies using a DHCP option code 51 this value is used if it is not less than the minimum (in this case the minimum is returned) or greater than the maximum (in this case the maximum is used).

Note Both **renew-timer** and **rebind-timer** are optional. The server will only send rebind-timer to the client, via DHCPv4 option code 59, if it is less than lease valid lifetime; and it will only send renew-timer, via DHCPv4 option code 58, if it is less than rebind-timer (or lease valid lifetime if rebind-timer was not specified). In their absence, the client should select values for T1 and T2 timers according to [RFC 2131](#). See section [Section 8.2.9](#) for more details on generating T1 and T2.

The **interfaces-config** map specifies the server configuration concerning the network interfaces, on which the server should listen to the DHCP messages. The **interfaces** parameter specifies a list of network interfaces on which the server should listen. Lists are opened and closed with square brackets, with elements separated by commas. To listen on two interfaces, the **interfaces-config** command should look like this:

```
"interfaces-config": {  
  "interfaces": [ "eth0", "eth1" ]  
},
```

The next couple of lines define the lease database, the place where the server stores its lease information. This particular example tells the server to use **memfile**, which is the simplest (and fastest) database backend. It uses an in-memory database and stores leases on disk in a CSV file. This is a very simple configuration; usually the lease database configuration is more extensive and contains additional parameters. Note that **lease-database** is an object and opens up a new scope, using an opening brace. Its parameters (just one in this example - **type**) follow. If there were more than one, they would be separated by commas. This scope is closed with a closing brace. As more parameters for the Dhcp4 definition follow, a trailing comma is present.

Finally, we need to define a list of IPv4 subnets. This is the most important DHCPv4 configuration structure, as the server uses that information to process clients' requests. It defines all subnets from which the server is expected to receive DHCP requests. The subnets are specified with the **subnet4** parameter. It is a list, so it starts and ends with square brackets. Each subnet definition in the list has several attributes associated with it, so it is a structure and is opened and closed with braces. At a minimum, a



subnet definition has to have at least two parameters: **subnet** (which defines the whole subnet) and **pools** (which is a list of dynamically allocated pools that are governed by the DHCP server).

The example contains a single subnet. If more than one were defined, additional elements in the **subnet4** parameter would be specified and separated by commas. For example, to define three subnets, the following syntax would be used:

```
"subnet4": [
  {
    "pools": [ { "pool": "192.0.2.1 - 192.0.2.200" } ],
    "subnet": "192.0.2.0/24"
  },
  {
    "pools": [ { "pool": "192.0.3.100 - 192.0.3.200" } ],
    "subnet": "192.0.3.0/24"
  },
  {
    "pools": [ { "pool": "192.0.4.1 - 192.0.4.254" } ],
    "subnet": "192.0.4.0/24"
  }
]
```

Note that indentation is optional and is used for aesthetic purposes only. In some cases it may be preferable to use more compact notation.

After all the parameters have been specified, we have two contexts open: global and Dhcp4, hence we need two closing curly brackets to close them.

8.2.2 Lease Storage

All leases issued by the server are stored in the lease database. Currently there are four database backends available: memfile (which is the default backend), MySQL, PostgreSQL, and Cassandra.

8.2.2.1 Memfile - Basic Storage for Leases

The server is able to store lease data in different repositories. Larger deployments may elect to store leases in a database. Section 8.2.2.2 describes this option. In typical smaller deployments though, the server will store lease information in a CSV file rather than a database. As well as requiring less administration, an advantage of using a file for storage is that it eliminates a dependency on third-party database software.

The configuration of the file backend (Memfile) is controlled through the Dhcp4/lease-database parameters. The **type** parameter is mandatory and it specifies which storage for leases the server should use. The value of "**memfile**" indicates that the file should be used as the storage. The following list gives additional optional parameters that can be used to configure the Memfile backend.

- **persist**: controls whether the new leases and updates to existing leases are written to the file. It is strongly recommended that the value of this parameter is set to **true** at all times during the server's normal operation. Not writing leases to disk means that if a server is restarted (e.g. after a power failure), it will not know what addresses have been assigned. As a result, it may hand out addresses to new clients that are already in use. The value of **false** is mostly useful for performance-testing purposes. The default value of the **persist** parameter is **true**, which enables writing lease updates to the lease file.
- **name**: specifies an absolute location of the lease file in which new leases and lease updates will be recorded. The default value for this parameter is "**[kea-install-dir]/var/lib/kea/kea-leases4.csv**".
- **lfc-interval**: specifies the interval, in seconds, at which the server will perform a lease file cleanup (LFC). This removes redundant (historical) information from the lease file and effectively reduces the lease file size. The cleanup process is described in more detailed fashion later in this section. The default value of the **lfc-interval** is **3600**. A value of 0 disables the LFC.

An example configuration of the Memfile backend is presented below:



```
"Dhcp4": {
  "lease-database": {
    "type": "memfile",
    "persist": true,
    "name": "/tmp/kea-leases4.csv",
    "lfc-interval": 1800
  }
}
```

This configuration selects the `/tmp/kea-leases4.csv` as the storage for lease information and enables persistence (writing lease updates to this file). It also configures the backend to perform a periodic cleanup of the lease file every 30 minutes.

It is important to know how the lease file contents are organized to understand why the periodic lease file cleanup is needed. Every time the server updates a lease or creates a new lease for the client, the new lease information must be recorded in the lease file. For performance reasons, the server does not update the existing client's lease in the file, as this would potentially require rewriting the entire file. Instead, it simply appends the new lease information to the end of the file; the previous lease entries for the client are not removed. When the server loads leases from the lease file, e.g. at the server startup, it assumes that the latest lease entry for the client is the valid one. The previous entries are discarded, meaning that the server can re-construct the accurate information about the leases even though there may be many lease entries for each client. However, storing many entries for each client results in a bloated lease file and impairs the performance of the server's startup and reconfiguration, as it needs to process a larger number of lease entries.

Lease file cleanup (LFC) removes all previous entries for each client and leaves only the latest ones. The interval at which the cleanup is performed is configurable, and it should be selected according to the frequency of lease renewals initiated by the clients. The more frequent the renewals, the smaller the value of **lfc-interval** should be. Note, however, that the LFC takes time and thus it is possible (although unlikely) that, if the **lfc-interval** is too short, a new cleanup may be started while the previous one is still running. The server would recover from this by skipping the new cleanup when it detects that the previous cleanup is still in progress. But it implies that the actual cleanups will be triggered more rarely than configured. Moreover, triggering a new cleanup adds overhead to the server, which will not be able to respond to new requests for a short period of time when the new cleanup process is spawned. Therefore, it is recommended that the **lfc-interval** value is selected in a way that would allow for the LFC to complete the cleanup before a new cleanup is triggered.

Lease file cleanup is performed by a separate process (in the background) to avoid a performance impact on the server process. To avoid the conflicts between two processes both using the same lease files, the LFC process starts with Kea opening new lease file and the actual LFC process operates on the lease file that is no longer used by the server. There are also other files created as a side effect of the lease file cleanup. The detailed description of the LFC is located later in this Kea Administrator's Reference Manual: Chapter 13.

8.2.2.2 Lease Database Configuration

Note

Lease database access information must be configured for the DHCPv4 server, even if it has already been configured for the DHCPv6 server. The servers store their information independently, so each server can use a separate database or both servers can use the same database.

Lease database configuration is controlled through the `Dhcp4/lease-database` parameters. The type of the database must be set to `"memfile"`, `"mysql"`, `"postgresql"`, or `"cql"`, e.g.:

```
"Dhcp4": { "lease-database": { "type": "mysql", ... }, ... }
```

Next, the name of the database to hold the leases must be set; this is the name used when the database was created (see Section 4.3.2.1, Section 4.3.3.1, or Section 4.3.4.1).

```
"Dhcp4": { "lease-database": { "name": "database-name" , ... }, ... }
```

For Cassandra:



```
"Dhcp4": { "lease-database": { "keyspace": "database-name" , ... }, ... }
```

If the database is located on a different system from the DHCPv4 server, the database host name must also be specified. (It should be noted that this configuration may have a severe impact on server performance.):

```
"Dhcp4": { "lease-database": { "host": "remote-host-name", ... }, ... }
```

Normally, the database will be on the same machine as the DHCPv4 server. In this case, set the value to the empty string:

```
"Dhcp4": { "lease-database": { "host" : "", ... }, ... }
```

Should the database use a port different than the default, it may be specified as well:

```
"Dhcp4": { "lease-database": { "port" : 12345, ... }, ... }
```

Should the database be located on a different system, you may need to specify a longer interval for the connection timeout:

```
"Dhcp4": { "lease-database": { "connect-timeout" : timeout-in-seconds, ... }, ... }
```

The default value of five seconds should be more than adequate for local connections. If a timeout is given, though, it should be an integer greater than zero.

The maximum number of times the server will automatically attempt to reconnect to the lease database after connectivity has been lost may be specified:

```
"Dhcp4": { "lease-database": { "max-reconnect-tries" : number-of-tries, ... }, ... }
```

If the server is unable to reconnect to the database after making the maximum number of attempts the server will exit. A value of zero (the default) disables automatic recovery and the server will exit immediately upon detecting a loss of connectivity (MySQL and Postgres only). For Cassandra, Kea uses a Cassandra interface that connects to all nodes in a cluster at the same time. Any connectivity issues should be handled by internal Cassandra mechanisms.

The number of milliseconds the server will wait between attempts to reconnect to the lease database after connectivity has been lost may also be specified:

```
"Dhcp4": { "lease-database": { "reconnect-wait-time" : number-of-milliseconds, ... }, ... }
```

The default value for MySQL and Postgres is 0, which disables automatic recovery and causes the server to exit immediately upon detecting the loss of connectivity. The default value for Cassandra is 2000 ms.

Note

Automatic reconnection to database backends is configured individually per backend. This allows you to tailor the recovery parameters to each backend you use. We do suggest that you enable it either for all backends or no backends so you have consistent behavior. Losing connectivity to a backend for which reconnect is disabled will result in the server shutting itself down. This includes cases when the lease database backend and the hosts database backend are connected to the same database instance.

Note Note that host parameter is used by MySQL and PostgreSQL backends. Cassandra has a concept of contact points that could be used to contact the cluster, instead of a single IP or hostname. It takes a list of comma-separated IP addresses, which may be specified as:

```
"Dhcp4": { "lease-database": { "contact-points" : "192.0.2.1,192.0.2.2", ... }, ... }
```

Finally, the credentials of the account under which the server will access the database should be set:

```
"Dhcp4": { "lease-database": { "user": "user-name",
                             "password": "password",
                             ... },
  ... }
```

If there is no password to the account, set the password to the empty string "". (This is also the default.)



8.2.2.3 Cassandra-Specific Parameters

The Cassandra backend is configured slightly differently. Cassandra has a concept of contact points that could be used to contact the cluster, instead of a single IP or hostname. It takes a list of comma-separated IP addresses, which may be specified as:

```
"Dhcp4": {
  "lease-database": {
    "type": "cql",
    "contact-points": "ip-address1, ip-address2 [...]",
    ...
  },
  ...
}
```

Cassandra also supports a number of optional parameters:

- **reconnect-wait-time** - governs how long Kea waits before attempting to reconnect. Expressed in milliseconds. The default is 2000 [ms].
- **connect-timeout** - sets the timeout for connecting to a node. Expressed in milliseconds. The default is 5000 [ms].
- **request-timeout** - sets the timeout for waiting for a response from a node. Expressed in milliseconds. The default is 12000 [ms].
- **tcp-keepalive** - governs the TCP keep-alive mechanism. Expressed in seconds of delay. If the parameter is not present, the mechanism is disabled.
- **tcp-nodelay** - enables/disables Nagle's algorithm on connections. The default is true.
- **consistency** - configures consistency level. The default is "quorum". Supported values: any, one, two, three, quorum, all, local-quorum, each-quorum, serial, local-serial, local-one. See [Cassandra consistency](#) for more details.
- **serial-consistency** - configures serial consistency level which manages lightweight transaction isolation. The default is "serial". Supported values: any, one, two, three, quorum, all, local-quorum, each-quorum, serial, local-serial, local-one. See [Cassandra serial consistency](#) for more details.

For example, a complex Cassandra configuration with most parameters specified could look as follows:

```
"Dhcp4": {
  "lease-database": {
    "type": "cql",
    "keyspace": "keatest",
    "contact-points": "192.0.2.1, 192.0.2.2, 192.0.2.3",
    "port": 9042,
    "reconnect-wait-time": 2000,
    "connect-timeout": 5000,
    "request-timeout": 12000,
    "tcp-keepalive": 1,
    "tcp-nodelay": true
  },
  ...
}
```

Similar parameters can be specified for the hosts database.

8.2.3 Hosts Storage

Kea is also able to store information about host reservations in the database. The hosts database configuration uses the same syntax as the lease database. In fact, a Kea server opens independent connections for each purpose, be it lease or hosts information.



This arrangement gives the most flexibility. Kea can keep leases and host reservations separately, but can also point to the same database. Currently the supported hosts database types are MySQL, PostgreSQL, and Cassandra.

Please note that usage of hosts storage is optional. A user can define all host reservations in the configuration file, and that is the recommended way if the number of reservations is small. However, when the number of reservations grows, it is more convenient to use host storage. Please note that both storage methods (configuration file and one of the supported databases) can be used together. If hosts are defined in both places, the definitions from the configuration file are checked first and external storage is checked later, if necessary.

In fact, host information can be placed in multiple stores. Operations are performed on the stores in the order they are defined in the configuration file, although this leads to a restriction in ordering in the case of a host reservation addition; read-only stores must be configured after a (required) read-write store, or the addition will fail.

8.2.3.1 DHCPv4 Hosts Database Configuration

Hosts database configuration is controlled through the `Dhcp4/hosts-database` parameters. If enabled, the type of database must be set to `"mysql"` or `"postgresql"`.

```
"Dhcp4": { "hosts-database": { "type": "mysql", ... }, ... }
```

Next, the name of the database to hold the reservations must be set; this is the name used when the lease database was created (see Section 4.3 for instructions on how to set up the desired database type).

```
"Dhcp4": { "hosts-database": { "name": "database-name", ... }, ... }
```

If the database is located on a different system than the DHCPv4 server, the database host name must also be specified. (Again it should be noted that this configuration may have a severe impact on server performance.)

```
"Dhcp4": { "hosts-database": { "host": remote-host-name, ... }, ... }
```

Normally, the database will be on the same machine as the DHCPv4 server. In this case, set the value to the empty string:

```
"Dhcp4": { "hosts-database": { "host": "", ... }, ... }
```

Should the database use a port different than the default, it may be specified as well:

```
"Dhcp4": { "hosts-database": { "port": 12345, ... }, ... }
```

The maximum number of times the server will automatically attempt to reconnect to the host database after connectivity has been lost may be specified:

```
"Dhcp4": { "hosts-database": { "max-reconnect-tries": number-of-tries, ... }, ... }
```

If the server is unable to reconnect to the database after making the maximum number of attempts the server will exit. A value of zero (the default) disables automatic recovery and the server will exit immediately upon detecting a loss of connectivity (MySQL and Postgres only).

The number of milliseconds the server will wait between attempts to reconnect to the host database after connectivity has been lost may also be specified:

```
"Dhcp4": { "hosts-database": { "reconnect-wait-time": number-of-milliseconds, ... }, ... }
```

The default value for MySQL and Postgres is 0, which disables automatic recovery and causes the server to exit immediately upon detecting the loss of connectivity. The default value for Cassandra is 2000 ms.

Note

Automatic reconnection to database backends is configured individually per backend. This allows you to tailor the recovery parameters to each backend you use. We do suggest that you enable it either for all backends or no backends so you have consistent behavior. Losing connectivity to a backend for which reconnect is disabled will result in the server shutting itself down. This includes cases when the lease database backend and the hosts database backend are connected to the same database instance.



Finally, the credentials of the account under which the server will access the database should be set:

```
"Dhcp4": { "hosts-database": { "user": "user-name",
                              "password": "password",
                              ... },
          ... }
```

If there is no password to the account, set the password to the empty string "". (This is also the default.)

The multiple storage extension uses a similar syntax; a configuration is placed into a "hosts-databases" list instead of into a "hosts-database" entry as in:

```
"Dhcp4": { "hosts-databases": [ { "type": "mysql", ... }, ... ], ... }
```

For additional Cassandra-specific parameters, see Section [8.2.2.3](#).

8.2.3.2 Using Read-Only Databases for Host Reservations

In some deployments the database user whose name is specified in the database backend configuration may not have write privileges to the database. This is often required by the policy within a given network to secure the data from being unintentionally modified. In many cases administrators have deployed inventory databases, which contain substantially more information about the hosts than just the static reservations assigned to them. The inventory database can be used to create a view of a Kea hosts database and such a view is often read-only.

Kea host database backends operate with an implicit configuration to both read from and write to the database. If the database user does not have write access to the host database, the backend will fail to start and the server will refuse to start (or reconfigure). However, if access to a read-only host database is required for retrieving reservations for clients and/or assigning specific addresses and options, it is possible to explicitly configure Kea to start in "read-only" mode. This is controlled by the **readonly** boolean parameter as follows:

```
"Dhcp4": { "hosts-database": { "readonly": true, ... }, ... }
```

Setting this parameter to **false** configures the database backend to operate in "read-write" mode, which is also the default configuration if the parameter is not specified.

Note

The **readonly** parameter is currently only supported for MySQL and PostgreSQL databases.

8.2.4 Interface Configuration

The DHCPv4 server must be configured to listen on specific network interfaces. The simplest network interface configuration tells the server to listen on all available interfaces:

```
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "*" ]
  }
  ...
},
```

The asterisk plays the role of a wildcard and means "listen on all interfaces." However, it is usually a good idea to explicitly specify interface names:

```
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "eth1", "eth3" ]
  },
  ...
}
```



It is possible to use a wildcard interface name (asterisk) concurrently with explicit interface names:

```
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "eth1", "eth3", "*" ]
  },
  ...
}
```

It is anticipated that this form of usage will only be used when it is desired to temporarily override a list of interface names and listen on all interfaces.

Some deployments of DHCP servers require that the servers listen on interfaces with multiple IPv4 addresses configured. In these situations, the address to use can be selected by appending an IPv4 address to the interface name in the following manner:

```
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "eth1/10.0.0.1", "eth3/192.0.2.3" ]
  },
  ...
}
```

Should the server be required to listen on multiple IPv4 addresses assigned to the same interface, multiple addresses can be specified for an interface as in the example below:

```
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "eth1/10.0.0.1", "eth1/10.0.0.2" ]
  },
  ...
}
```

Alternatively, if the server should listen on all addresses for the particular interface, an interface name without any address should be specified.

Kea supports responding to directly connected clients which don't have an address configured. This requires the server to inject the hardware address of the destination into the data link layer of the packet being sent to the client. The DHCPv4 server uses raw sockets to achieve this, and builds the entire IP/UDP stack for the outgoing packets. The downside of raw socket use, however, is that incoming and outgoing packets bypass the firewalls (e.g. iptables). It is also troublesome to handle traffic on multiple IPv4 addresses assigned to the same interface, as raw sockets are bound to the interface; plus, advanced packet filtering techniques (e.g. using the BPF) have to be used to receive unicast traffic on the desired addresses assigned to the interface, rather than capturing whole traffic reaching the interface to which the raw socket is bound. Therefore, in deployments where the server doesn't have to provision the directly connected clients and only receives the unicast packets from the relay agents, the DHCP server should be configured to use IP/UDP datagram sockets instead of raw sockets. The following configuration demonstrates how this can be achieved:

```
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "eth1", "eth3" ],
    "dhcp-socket-type": "udp"
  },
  ...
}
```

The **dhcp-socket-type** specifies that the IP/UDP sockets will be opened on all interfaces on which the server listens, i.e. "eth1" and "eth3" in our case. If **dhcp-socket-type** is set to **raw**, it configures the server to use raw sockets instead. If the **dhcp-socket-type** value is not specified, the default value **raw** is used.

Using UDP sockets automatically disables the reception of broadcast packets from directly connected clients. This effectively means that UDP sockets can be used for relayed traffic only. When using raw sockets, both the traffic from the directly connected clients and the relayed traffic are handled. Caution should be taken when configuring the server to open multiple raw sockets on



the interface with several IPv4 addresses assigned. If the directly connected client sends the message to the broadcast address, all sockets on this link will receive this message and multiple responses will be sent to the client. Therefore, the configuration with multiple IPv4 addresses assigned to the interface should not be used when the directly connected clients are operating on that link. To use a single address on such interface, the "interface-name/address" notation should be used.

Note

Specifying the value **raw** as the socket type doesn't guarantee that the raw sockets will be used! The use of raw sockets to handle the traffic from the directly connected clients is currently supported on Linux and BSD systems only. If the raw sockets are not supported on your particular OS, the server will issue a warning and fall back to using IP/UDP sockets.

In a typical environment, the DHCP server is expected to send back a response on the same network interface on which the query was received. This is the default behavior. However, in some deployments it is desired that the outbound (response) packets will be sent as regular traffic and the outbound interface will be determined by the routing tables. This kind of asymmetric traffic is uncommon, but valid. Kea now supports a parameter called **outbound-interface** that controls this behavior. It supports two values. The first one, **same-as-inbound**, tells Kea to send back the response on the same interface where the query packet was received. This is the default behavior. The second one, **use-routing**, tells Kea to send regular UDP packets and let the kernel's routing table determine the most appropriate interface. This only works when **dhcp-socket-type** is set to **udp**. An example configuration looks as follows:

```
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "eth1", "eth3" ],
    "dhcp-socket-type": "udp",
    "outbound-interface": "use-routing"
  },
  ...
}
```

Interfaces are re-detected at each reconfiguration. This behavior can be disabled by setting the **re-detect** value to **false**, for instance:

```
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "eth1", "eth3" ],
    "re-detect": false
  },
  ...
}
```

Note that interfaces are not re-detected during **config-test**.

Usually loopback interfaces (e.g. the "lo" or "lo0" interface) may not be configured, but if a loopback interface is explicitly configured and IP/UDP sockets are specified, the loopback interface is accepted.

For example, it can be used to run Kea in a FreeBSD jail having only a loopback interface, to service a relayed DHCP request:

```
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "lo0" ],
    "dhcp-socket-type": "udp"
  },
  ...
}
```

8.2.5 Issues with Unicast Responses to DHCPINFORM

The use of UDP sockets has certain benefits in deployments where the server receives only relayed traffic; these benefits are mentioned in Section 8.2.4. From the administrator's perspective it is often desirable to configure the system's firewall to filter



out the unwanted traffic, and the use of UDP sockets facilitates this. However, the administrator must also be aware of the implications related to filtering certain types of traffic, as it may impair the DHCP server's operation.

In this section we are focusing on the case when the server receives the DHCPINFORM message from the client via a relay. According to [RFC 2131](#), the server should unicast the DHCPACK response to the address carried in the "ciaddr" field. When the UDP socket is in use, the DHCP server relies on the low-level functions of an operating system to build the data link, IP, and UDP layers of the outgoing message. Typically, the OS will first use ARP to obtain the client's link-layer address to be inserted into the frame's header, if the address is not cached from a previous transaction that the client had with the server. When the ARP exchange is successful, the DHCP message can be unicast to the client, using the obtained address.

Some system administrators block ARP messages in their network, which causes issues for the server when it responds to the DHCPINFORM messages, because the server is unable to send the DHCPACK if the preceding ARP communication fails. Since the OS is entirely responsible for the ARP communication and then sending the DHCP packet over the wire, the DHCP server has no means to determine that the ARP exchange failed and the DHCP response message was dropped. Thus, the server does not log any error messages when the outgoing DHCP response is dropped. At the same time, all hooks pertaining to the packet-sending operation will be called, even though the message never reaches its destination.

Note that the issue described in this section is not observed when the raw sockets are in use, because, in this case, the DHCP server builds all the layers of the outgoing message on its own and does not use ARP. Instead, it inserts the value carried in the 'chaddr' field of the DHCPINFORM message into the link layer.

Server administrators willing to support DHCPINFORM messages via relays should not block ARP traffic in their networks or should use raw sockets instead of UDP sockets.

8.2.6 IPv4 Subnet Identifier

The subnet identifier is a unique number associated with a particular subnet. In principle, it is used to associate clients' leases with their respective subnets. When a subnet identifier is not specified for a subnet being configured, it will be automatically assigned by the configuration mechanism. The identifiers are assigned from 1 and are monotonically increased for each subsequent subnet: 1, 2, 3

If there are multiple subnets configured with auto-generated identifiers and one of them is removed, the subnet identifiers may be renumbered. For example: if there are four subnets and the third is removed, the last subnet will be assigned the identifier that the third subnet had before removal. As a result, the leases stored in the lease database for subnet 3 are now associated with subnet 4, something that may have unexpected consequences. The only remedy for this issue at present is to manually specify a unique identifier for each subnet.

Note Subnet IDs must be greater than zero and less than 4294967295.

The following configuration will assign the specified subnet identifier to the newly configured subnet:

```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "id": 1024,
      ...
    }
  ]
}
```

This identifier will not change for this subnet unless the "id" parameter is removed or set to 0. The value of 0 forces auto-generation of the subnet identifier.

8.2.7 IPv4 Subnet Prefix

The subnet prefix is the second way to identify a subnet. It does not need to have the address part to match the prefix length, for instance this configuration is accepted:



```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.1/24",
      ...
    }
  ]
}
```

Even there is another subnet with the "192.0.2.0/24" prefix: only the textual form of subnets are compared to avoid duplicates.

Note Abuse of this feature can lead to incorrect subnet selection (see Section 8.6).

8.2.8 Configuration of IPv4 Address Pools

The main role of a DHCPv4 server is address assignment. For this, the server must be configured with at least one subnet and one pool of dynamic addresses to be managed. For example, assume that the server is connected to a network segment that uses the 192.0.2.0/24 prefix. The administrator of that network decides that addresses from range 192.0.2.10 to 192.0.2.20 are going to be managed by the Dhcp4 server. Such a configuration can be achieved in the following way:

```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [
        { "pool": "192.0.2.10 - 192.0.2.20" }
      ],
      ...
    }
  ]
}
```

Note that **subnet** is defined as a simple string, but the **pools** parameter is actually a list of pools; for this reason, the pool definition is enclosed in square brackets, even though only one range of addresses is specified.

Each **pool** is a structure that contains the parameters that describe a single pool. Currently there is only one parameter, **pool**, which gives the range of addresses in the pool.

It is possible to define more than one pool in a subnet; continuing the previous example, further assume that 192.0.2.64/26 should be also be managed by the server. It could be written as 192.0.2.64 to 192.0.2.127. Alternatively, it can be expressed more simply as 192.0.2.64/26. Both formats are supported by Dhcp4 and can be mixed in the pool list. For example, one could define the following pools:

```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [
        { "pool": "192.0.2.10-192.0.2.20" },
        { "pool": "192.0.2.64/26" }
      ],
      ...
    }
  ],
  ...
}
```



White space in pool definitions is ignored, so spaces before and after the hyphen are optional. They can be used to improve readability.

The number of pools is not limited, but for performance reasons it is recommended to use as few as possible.

The server may be configured to serve more than one subnet:

```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [ { "pool": "192.0.2.1 - 192.0.2.200" } ],
      ...
    },
    {
      "subnet": "192.0.3.0/24",
      "pools": [ { "pool": "192.0.3.100 - 192.0.3.200" } ],
      ...
    },
    {
      "subnet": "192.0.4.0/24",
      "pools": [ { "pool": "192.0.4.1 - 192.0.4.254" } ],
      ...
    }
  ]
}
```

When configuring a DHCPv4 server using prefix/length notation, please pay attention to the boundary values. When specifying that the server can use a given pool, it will also be able to allocate the first (typically a network address) and the last (typically a broadcast address) address from that pool. In the aforementioned example of pool 192.0.3.0/24, both the 192.0.3.0 and 192.0.3.255 addresses may be assigned as well. This may be invalid in some network configurations. To avoid this, please use the "min-max" notation.

8.2.9 Sending T1 (Option 58) and T2 (Option 59)

According to [RFC 2131](#), servers should send values for T1 and T2 that are 50% and 87.5% of the lease life time, respectively. By default, kea-dhcp4 does not send either value. It can be configured to send values that are specified explicitly or that are calculated as percentages of the lease time. The server's behavior is governed by combination of configuration parameters, two of which have already been mentioned.

To send specific, fixed values use the following two parameters:

- **renew-timer** - specifies the value of T1 in seconds.
- **rebind-timer** - specifies the value of T2 in seconds.

The server will only send T2 if it is less than valid lease time. T1 will only be sent if a: T2 is being sent and T1 is less than T2 or b: T2 is not being sent and T1 is less than the valid lease time.

Calculating the values is controlled by the following three parameters.

- **calculate-tee-times** - when true, T1 and T2 will be calculated as percentages of the valid lease time. It defaults to false.
- **t1-percent** - the percentage of the valid lease time to use for T1. It is expressed as a real number between 0.0 and 1.0 and must be less than t2-percent. The default value is 0.50 per RFC 2131.
- **t2-percent** - the percentage of the valid lease time to use for T2. It is expressed as a real number between 0.0 and 1.0 and must be greater than t1-percent. The default value is .875 per RFC 2131.



Note In the event that both explicit values are specified and `calculate-tee-times` is true, the server will use the explicit values. If you plan on having a mixture where some subnets or share-networks will use explicit values and some will use calculated values you must not define the explicit values at any level higher than where they will be used. Inheriting them from too high of a scope, such as global, will cause them to have values at every level underneath (shared-networks and subnets), effectively disabling calculated values.

8.2.10 Standard DHCPv4 Options

One of the major features of the DHCPv4 server is to provide configuration options to clients. Most of the options are sent by the server only if the client explicitly requests them using the Parameter Request List option. Those that do not require inclusion in the Parameter Request List option are commonly used options, e.g. "Domain Server", and options which require special behavior, e.g. "Client FQDN", which is returned to the client if the client has included this option in its message to the server.

Table 8.1 comprises the list of the standard DHCPv4 options whose values can be configured using the configuration structures described in this section. This table excludes the options which require special processing and thus cannot be configured with some fixed values. The last column of the table indicates which options can be sent by the server even when they are not requested in the Parameter Request list option, and those which are sent only when explicitly requested.

The following example shows how to configure the addresses of DNS servers, which is one of the most frequently used options. Options specified in this way are considered global and apply to all configured subnets.

```
"Dhcp4": {
  "option-data": [
    {
      "name": "domain-name-servers",
      "code": 6,
      "space": "dhcp4",
      "csv-format": true,
      "data": "192.0.2.1, 192.0.2.2"
    },
    ...
  ]
}
```

Note that only one of name or code is required; you don't need to specify both. Space has a default value of "dhcp4", so you can skip this as well if you define a regular (not encapsulated) DHCPv4 option. Finally, `csv-format` defaults to true, so it too can be skipped, unless you want to specify the option value as a hexadecimal string. Therefore, the above example can be simplified to:

```
"Dhcp4": {
  "option-data": [
    {
      "name": "domain-name-servers",
      "data": "192.0.2.1, 192.0.2.2"
    },
    ...
  ]
}
```

Defined options are added to the response when the client requests them at a few exceptions, which are always added. To enforce the addition of a particular option set the `always-send` flag to true as in:

```
"Dhcp4": {
  "option-data": [
    {
      "name": "domain-name-servers",
      "data": "192.0.2.1, 192.0.2.2",
      "always-send": true
    },
    ...
  ]
}
```



```
]
}
```

The effect is the same as if the client added the option code in the Parameter Request List option (or its equivalent for vendor options):

```
"Dhcp4": {
  "option-data": [
    {
      "name": "domain-name-servers",
      "data": "192.0.2.1, 192.0.2.2",
      "always-send": true
    },
    ...
  ],
  "subnet4": [
    {
      "subnet": "192.0.3.0/24",
      "option-data": [
        {
          "name": "domain-name-servers",
          "data": "192.0.3.1, 192.0.3.2"
        },
        ...
      ],
      ...
    },
    ...
  ],
  ...
}
```

The Domain Name Servers option is always added to responses (the `always-send` is "sticky") but the value is the subnet one when the client is localized in the subnet.

The **name** parameter specifies the option name. For a list of currently supported names, see Table 8.1 below. The **code** parameter specifies the option code, which must match one of the values from that list. The next line specifies the option space, which must always be set to "dhcp4" as these are standard DHCPv4 options. For other option spaces, including custom option spaces, see Section 8.2.14. The next line specifies the format in which the data will be entered; use of CSV (comma-separated values) is recommended. The sixth line gives the actual value to be sent to clients. Data is specified as normal text, with values separated by commas if more than one value is allowed.

Options can also be configured as hexadecimal values. If **csv-format** is set to false, option data must be specified as a hexadecimal string. The following commands configure the domain-name-servers option for all subnets with the following addresses: 192.0.3.1 and 192.0.3.2. Note that **csv-format** is set to false.

```
"Dhcp4": {
  "option-data": [
    {
      "name": "domain-name-servers",
      "code": 6,
      "space": "dhcp4",
      "csv-format": false,
      "data": "C0 00 03 01 C0 00 03 02"
    },
    ...
  ],
  ...
}
```

Kea supports the following formats when specifying hexadecimal data:



- **Delimited octets** One or more octets separated by either colons or spaces (':' or ' '). While each octet may contain one or two digits, we strongly recommend always using two digits. Valid examples are "ab:cd:ef" and "ab cd ef".
- **String of digits** A continuous string of hexadecimal digits with or without a "0x" prefix. Valid examples are "0xabcdef" and "abcdef".

Care should be taken to use proper encoding when using hexadecimal format. Kea's ability to validate data correctness in hexadecimal is limited.

Most of the parameters in the "option-data" structure are optional and can be omitted in some circumstances as discussed in Section 8.2.15.

It is possible to specify or override options on a per-subnet basis. If clients connected to most of your subnets are expected to get the same values of a given option, you should use global options; you can then override specific values for a small number of subnets. On the other hand, if you use different values in each subnet, it does not make sense to specify global option values; rather, you should set only subnet-specific ones.

The following commands override the global DNS servers option for a particular subnet, setting a single DNS server with address 192.0.2.3:

```
"Dhcp4": {
  "subnet4": [
    {
      "option-data": [
        {
          "name": "domain-name-servers",
          "code": 6,
          "space": "dhcp4",
          "csv-format": true,
          "data": "192.0.2.3"
        },
        ...
      ],
      ...
    },
    ...
  ],
  ...
}
```

In some cases it is useful to associate some options with an address pool from which a client is assigned a lease. Pool-specific option values override subnet-specific and global option values. The server's administrator must not try to prioritize assignment of pool-specific options by trying to order pool declarations in the server configuration.

The following configuration snippet demonstrates how to specify the DNS servers option, which will be assigned to a client only if the client obtains an address from the given pool:

```
"Dhcp4": {
  "subnet4": [
    {
      "pools": [
        {
          "pool": "192.0.2.1 - 192.0.2.200",
          "option-data": [
            {
              "name": "domain-name-servers",
              "data": "192.0.2.3"
            },
            ...
          ],
          ...
        },
        ...
      ],
      ...
    },
    ...
  ],
  ...
}
```



```
    ],  
    ...  
  },  
  ...  
],  
...  
}
```

Options can also be specified in class or host reservation scope. The current Kea options precedence order is (from most important): host reservation, pool, subnet, shared network, class, global.

The currently supported standard DHCPv4 options are listed in Table 8.1. "Name" and "Code" are the values that should be used as a name/code in the option-data structures. "Type" designates the format of the data; the meanings of the various types is given in Table 8.2.

When a data field is a string and that string contains the comma (,; U+002C) character, the comma must be escaped with two backslashes (\; U+005C). This double escape is required because both the routine splitting CSV data into fields and JSON use the same escape character; a single escape (\,) would make the JSON invalid. For example, the string "foo,bar" would be represented as:

```
"Dhcp4": {  
  "subnet4": [  
    {  
      "pools": [  
        {  
          "option-data": [  
            {  
              "name": "boot-file-name",  
              "data": "foo\\,bar"  
            }  
          ]  
        },  
        ...  
      ],  
      ...  
    },  
    ...  
  ],  
  ...  
},  
...  
}
```

Some options are designated as arrays, which means that more than one value is allowed in such an option. For example, the option `time-servers` allows the specification of more than one IPv4 address, enabling clients to obtain the addresses of multiple NTP servers.

Section 8.2.11 describes the configuration syntax to create custom option definitions (formats). Creation of custom definitions for standard options is generally not permitted, even if the definition being created matches the actual option format defined in the RFCs. There is an exception to this rule for standard options for which Kea currently does not provide a definition. In order to use such options, a server administrator must create a definition as described in Section 8.2.11 in the 'dhcp4' option space. This definition should match the option format described in the relevant RFC, but the configuration mechanism will allow any option format as it currently has no means to validate it.

8.2.11 Custom DHCPv4 Options

Kea supports custom (non-standard) DHCPv4 options. Assume that we want to define a new DHCPv4 option called "foo" which will have a code 222 and will convey a single, unsigned, 32-bit integer value. We can define such an option by using the following entry in the configuration file:

```
"Dhcp4": {  
  "option-def": [  
    {  
      "code": 222,  
      "type": "uint32"  
    }  
  ]  
}
```



Name	Code	Type	Array?	Returned if not requested?
time-offset	2	int32	false	false
routers	3	ipv4-address	true	true
time-servers	4	ipv4-address	true	false
name-servers	5	ipv4-address	true	false
domain-name-servers	6	ipv4-address	true	true
log-servers	7	ipv4-address	true	false
cookie-servers	8	ipv4-address	true	false
lpr-servers	9	ipv4-address	true	false
impress-servers	10	ipv4-address	true	false
resource-location-servers	11	ipv4-address	true	false
boot-size	13	uint16	false	false
merit-dump	14	string	false	false
domain-name	15	fqdn	false	true
swap-server	16	ipv4-address	false	false
root-path	17	string	false	false
extensions-path	18	string	false	false
ip-forwarding	19	boolean	false	false
non-local-source-routing	20	boolean	false	false
policy-filter	21	ipv4-address	true	false
max-dgram-reassembly	22	uint16	false	false
default-ip-ttl	23	uint8	false	false
path-mtu-aging-timeout	24	uint32	false	false
path-mtu-plateau-table	25	uint16	true	false
interface-mtu	26	uint16	false	false
all-subnets-local	27	boolean	false	false
broadcast-address	28	ipv4-address	false	false
perform-mask-discovery	29	boolean	false	false
mask-supplier	30	boolean	false	false
router-discovery	31	boolean	false	false
router-solicitation-address	32	ipv4-address	false	false
static-routes	33	ipv4-address	true	false
trailer-encapsulation	34	boolean	false	false
arp-cache-timeout	35	uint32	false	false
ieee802-3-encapsulation	36	boolean	false	false
default-tcp-ttl	37	uint8	false	false
tcp-keepalive-interval	38	uint32	false	false
tcp-keepalive-garbage	39	boolean	false	false
nis-domain	40	string	false	false
nis-servers	41	ipv4-address	true	false
ntp-servers	42	ipv4-address	true	false
vendor-encapsulated-options	43	empty	false	false
netbios-name-servers	44	ipv4-address	true	false
netbios-dd-server	45	ipv4-address	true	false
netbios-node-type	46	uint8	false	false
netbios-scope	47	string	false	false
font-servers	48	ipv4-address	true	false
x-display-manager	49	ipv4-address	true	false
dhcp-option-overload	52	uint8	false	false
dhcp-server-identifier	54	ipv4-address	false	true
dhcp-message	56	string	false	false
dhcp-max-message-size	57	uint16	false	false



Name	Meaning
binary	An arbitrary string of bytes, specified as a set of hexadecimal digits.
boolean	A boolean value with allowed values true or false.
empty	No value; data is carried in suboptions.
fqdn	Fully qualified domain name (e.g. www.example.com).
ipv4-address	IPv4 address in the usual dotted-decimal notation (e.g. 192.0.2.1).
ipv6-address	IPv6 address in the usual colon notation (e.g. 2001:db8::1).
ipv6-prefix	IPv6 prefix and prefix length specified using CIDR notation, e.g. 2001:db8:1::/64. This data type is used to represent an 8-bit field conveying a prefix length and the variable length prefix value.
psid	PSID and PSID length separated by a slash, e.g. 3/4 specifies PSID=3 and PSID length=4. In the wire format it is represented by an 8-bit field carrying PSID length (in this case equal to 4) and the 16-bits-long PSID value field (in this case equal to "0011000000000000b" using binary notation). Allowed values for a PSID length are 0 to 16. See RFC 7597 for details about the PSID wire representation.
record	Structured data that may be comprised of any types (except "record" and "empty"). The array flag applies to the last field only.
string	Any text. Please note that Kea will silently discard any terminating/trailing nulls from the end of 'string' options when unpacking received packets. This is keeping with RFC 2132, Section 2
tuple	A length encoded as an 8- (16- for DHCPv6) bit unsigned integer followed by a string of this length.
uint8	8-bit unsigned integer with allowed values 0 to 255.
uint16	16-bit unsigned integer with allowed values 0 to 65535.
uint32	32-bit unsigned integer with allowed values 0 to 4294967295.
int8	8-bit signed integer with allowed values -128 to 127.
int16	16-bit signed integer with allowed values -32768 to 32767.
int32	32-bit signed integer with allowed values -2147483648 to 2147483647.

Table 8.2: List of Standard DHCP Option Types



```
{
  "name": "foo",
  "code": 222,
  "type": "uint32",
  "array": false,
  "record-types": "",
  "space": "dhcp4",
  "encapsulate": ""
}, ...
],
...
}
```

The **false** value of the **array** parameter determines that the option does NOT comprise an array of "uint32" values but is, instead, a single value. Two other parameters have been left blank: **record-types** and **encapsulate**. The former specifies the comma-separated list of option data fields, if the option comprises a record of data fields. The **record-types** value should be non-empty if **type** is set to "record"; otherwise it must be left blank. The latter parameter specifies the name of the option space being encapsulated by the particular option. If the particular option does not encapsulate any option space, it should be left blank. Note that the **option-def** configuration statement only defines the format of an option and does not set its value(s).

The **name**, **code**, and **type** parameters are required; all others are optional. The **array** default value is **false**. The **record-types** and **encapsulate** default values are blank (i.e. ""). The default **space** is "dhcp4".

Once the new option format is defined, its value is set in the same way as for a standard option. For example, the following commands set a global value that applies to all subnets.

```
"Dhcp4": {
  "option-data": [
    {
      "name": "foo",
      "code": 222,
      "space": "dhcp4",
      "csv-format": true,
      "data": "12345"
    }, ...
  ],
  ...
}
```

New options can take more complex forms than simple use of primitives (uint8, string, ipv4-address, etc); it is possible to define an option comprising a number of existing primitives.

For example, assume we want to define a new option that will consist of an IPv4 address, followed by an unsigned 16-bit integer, followed by a boolean value, followed by a text string. Such an option could be defined in the following way:

```
"Dhcp4": {
  "option-def": [
    {
      "name": "bar",
      "code": 223,
      "space": "dhcp4",
      "type": "record",
      "array": false,
      "record-types": "ipv4-address, uint16, boolean, string",
      "encapsulate": ""
    }, ...
  ],
  ...
}
```

The **type** is set to "record" to indicate that the option contains multiple values of different types. These types are given as a comma-separated list in the **record-types** field and should be ones from those listed in Table 8.2.



The values of the option are set as follows:

```
"Dhcp4": {
  "option-data": [
    {
      "name": "bar",
      "space": "dhcp4",
      "code": 223,
      "csv-format": true,
      "data": "192.0.2.100, 123, true, Hello World"
    }
  ],
  ...
}
```

csv-format is set to **true** to indicate that the **data** field comprises a command-separated list of values. The values in **data** must correspond to the types set in the **record-types** field of the option definition.

When **array** is set to **true** and **type** is set to "record", the last field is an array, i.e. it can contain more than one value, as in:

```
"Dhcp4": {
  "option-def": [
    {
      "name": "bar",
      "code": 223,
      "space": "dhcp4",
      "type": "record",
      "array": true,
      "record-types": "ipv4-address, uint16",
      "encapsulate": ""
    }, ...
  ],
  ...
}
```

The new option content is one IPv4 address followed by one or more 16-bit unsigned integers.

Note

In general, boolean values are specified as **true** or **false**, without quotes. Some specific boolean parameters may accept also **"true"**, **"false"**, **0**, **1**, **"0"**, and **"1"**.

Note

Numbers can be specified in decimal or hexadecimal format. The hexadecimal format can be either plain (e.g. abcd) or prefixed with 0x (e.g. 0xabcd).

8.2.12 DHCPv4 Private Options

Options with a code between 224 and 254 are reserved for private use. They can be defined at the global scope or at the client-class local scope; this allows option definitions to be used depending on context and option data to be set accordingly. For instance, to configure an old PXEClient vendor:

```
"Dhcp4": {
  "client-classes": [
    {
      "name": "pxeclient",
      "test": "option[vendor-class-identifier].text == 'PXEClient'",
      "option-def": [
```



```
        {
            "name": "configfile",
            "code": 209,
            "type": "string"
        }
    ],
    ...
}, ...
],
...
}
```

As the Vendor-Specific Information option (code 43) has vendor-specific format, i.e. can carry either raw binary value or sub-options, this mechanism is available for this option too.

In the following example taken from a real configuration, two vendor classes use the option 43 for different and incompatible purposes:

```
"Dhcp4": {
  "option-def": [
    {
      "name": "cookie",
      "code": 1,
      "type": "string",
      "space": "APC"
    },
    {
      "name": "mtftp-ip",
      "code": 1,
      "type": "ipv4-address",
      "space": "PXE"
    },
    ...
  ],
  "client-classes": [
    {
      "name": "APC",
      "test": "(option[vendor-class-identifier].text == 'APC'",
      "option-def": [
        {
          "name": "vendor-encapsulated-options",
          "type": "empty",
          "encapsulate": "APC"
        }
      ],
      "option-data": [
        {
          "name": "cookie",
          "space": "APC",
          "data": "1APC"
        },
        {
          "name": "vendor-encapsulated-options"
        },
        ...
      ],
      ...
    },
    {
      "name": "PXE",
      "test": "(option[vendor-class-identifier].text == 'PXE'",
      "option-def": [
```



```
        {
            "name": "vendor-encapsulated-options",
            "type": "empty",
            "encapsulate": "PXE"
        }
    ],
    "option-data": [
        {
            "name": "mtftp-ip",
            "space": "PXE",
            "data": "0.0.0.0"
        },
        {
            "name": "vendor-encapsulated-options"
        },
        ...
    ],
    ...
},
...
],
...
}
```

The definition used to decode a VSI option is:

1. The local definition of a client class the incoming packet belongs to;
2. If none, the global definition;
3. If none, the last-resort definition described in the next section [Section 8.2.13](#) (backwards-compatible with previous Kea versions).

Note

This last-resort definition for the Vendor-Specific Information option (code 43) is not compatible with a raw binary value. So when there are some known cases where a raw binary value will be used, a client class must be defined with a classification expression matching these cases and an option definition for the VSI option with a binary type and no encapsulation.

Note

Option definitions in client classes are allowed only for this limited option set (codes 43 and from 224 to 254), and only for DHCPv4.

8.2.13 DHCPv4 Vendor-Specific Options

Currently there are two option spaces defined for the DHCPv4 daemon: "dhcp4" (for the top-level DHCPv4 options) and "vendor-encapsulated-options-space", which is empty by default but in which options can be defined. Such options will be carried in the Vendor-Specific Information option (code 43). The following examples show how to define an option "foo" in that space that has a code 1, and comprises an IPv4 address, an unsigned 16-bit integer, and a string. The "foo" option is conveyed in a Vendor-Specific Information option.

The first step is to define the format of the option:

```
"Dhcp4": {
    "option-def": [
        {
            "name": "foo",
            "code": 1,
```



```
        "space": "vendor-encapsulated-options-space",
        "type": "record",
        "array": false,
        "record-types": "ipv4-address, uint16, string",
        "encapsulate": ""
    }
],
...
}
```

(Note that the option space is set to "vendor-encapsulated-options-space".) Once the option format is defined, the next step is to define actual values for that option:

```
"Dhcp4": {
  "option-data": [
    {
      "name": "foo",
      "space": "vendor-encapsulated-options-space",
      "code": 1,
      "csv-format": true,
      "data": "192.0.2.3, 123, Hello World"
    }
  ],
  ...
}
```

We also include the Vendor-Specific Information option, the option that conveys our sub-option "foo". This is required; otherwise the option will not be included in messages sent to the client.

```
"Dhcp4": {
  "option-data": [
    {
      "name": "vendor-encapsulated-options"
    }
  ],
  ...
}
```

Alternatively, the option can be specified using its code.

```
"Dhcp4": {
  "option-data": [
    {
      "code": 43
    }
  ],
  ...
}
```

Another popular option that is often somewhat imprecisely called vendor option is option 125. It's proper name is vendor-independent vendor-specific information option or vivso. The idea behind those options is that each vendor has its own unique set of options with their own custom formats. The vendor is identified by a 32 unsigned integer called enterprise-id or vendor-id. For example, vivso with vendor-id 4491 represents DOCSIS options and you are likely to see many of them when dealing with cable modems.

In Kea each vendor is represented by its own vendor space. Since there are hundreds of vendors and sometimes they use different option definitions for different hardware, it's impossible for Kea to support them all out of the box. Fortunately, it's easy to define support for new vendor options. Let's take an example of Genexis home gateway. This device requires sending vivso 125 option with a suboption 2 that contains a string with TFTP server URL. To support such a device, three steps are needed. First, we need to define option definitions that will explain how the option is supposed to be formed. Second, we will need to define option values. Third, we will need to tell Kea when to send those specific options. This last step will be done with client classification.

An example snippet of a configuration could look similar to the following:



```
{
  // First, we need to define that suboption 2 in vivso option for
  // vendor-id 25167 has specific format (it's a plain string in this example).
  // After this definition, we can specify values for option tftp.
  "option-def": [
    {
      // We define a short name, so the option could be referenced by name.
      // The option has code 2 and resides with vendor space 25167.
      // Its data is a plain string.
      "name": "tftp",
      "code": 2,
      "space": "vendor-25167",
      "type": "string"
    }
  ],

  "client-classes": [
    {
      // We now need to tell Kea how to recognize when to use vendor space 25167.
      // Usually we can use simple expression such as checking if the device
      // sent a vivso option with specific vendor-id, e.g. "vendor[4491].exists"
      // Unfortunately, Genexis is a bit unusual in this aspect, because it
      // doesn't send vivso. In this case we need to look into vendor class
      // (option code 60) and see if there's specific string that identifies
      // the device.
      "name": "cpe_genexis",
      "test": "substring(option[60].hex,0,7) == 'HMC1000'",

      // Once the device is recognized, we want to send two options:
      // the VIVSO option with vendor-id set to 25167 and a suboption 2.
      "option-data": [
        {
          "name": "vivso-suboptions",
          "data": "25167",
          "encapsulate": "vendor-25167"
        },

        // The suboption 2 value is defined as any other option. However,
        // we want to send this suboption 2, even when the client didn't
        // explicitly requested it (often there is no way to do that for
        // vendor options). Therefore we use always-send to force Kea
        // to always send this option when 25167 vendor space is involved.
        {
          "name": "tftp",
          "space": "vendor-25167",
          "data": "tftp://192.0.2.1/genexis/HMC1000.v1.3.0-R.img",
          "always-send": true
        }
      ]
    }
  ]
}
```

One aspect requires a bit broader comment. By default Kea sends back only those options that are requested by a client, unless there are protocol rules that tell DHCP server to always send an option. This approach works nicely for most cases and avoids problems with clients refusing responses with options they don't understand. Unfortunately, this is more blurry when we consider vendor options. Some vendors (such as docsis, identified by vendor options 4491) have a mechanism to request specific vendor options and Kea is able to honor that. Unfortunately, for many other vendors, such as Genexis (25167) discussed here, Kea does not have such a mechanism, so it can't send any suboptions on its own. To solve this issue, we came up with a concept of persistent options. Kea can be told to always send options, even if client didn't request them. This can be achieved by adding "always-send": true to your option definition. Note that in this particular case an option is defined in a vendor space 25167. With the "always-send" enabled, the option will be sent every time there is a need to deal with vendor space 25167.



Another possibility is to redefine the option; see Section [8.2.12](#).

8.2.14 Nested DHCPv4 Options (Custom Option Spaces)

It is sometimes useful to define a completely new option space. This is the case when a user creates a new option in the standard option space ("dhcp4") and wants this option to convey sub-options. Since they are in a separate space, sub-option codes will have a separate numbering scheme and may overlap with the codes of standard options.

Note that the creation of a new option space is not required when defining sub-options for a standard option, because it is created by default if the standard option is meant to convey any sub-options (see Section [8.2.13](#)).

Assume that we want to have a DHCPv4 option called "container" with code 222 that conveys two sub-options with codes 1 and 2. First we need to define the new sub-options:

```
"Dhcp4": {
  "option-def": [
    {
      "name": "subopt1",
      "code": 1,
      "space": "isc",
      "type": "ipv4-address",
      "record-types": "",
      "array": false,
      "encapsulate": ""
    },
    {
      "name": "subopt2",
      "code": 2,
      "space": "isc",
      "type": "string",
      "record-types": "",
      "array": false,
      "encapsulate": ""
    }
  ],
  ...
}
```

Note that we have defined the options to belong to a new option space (in this case, "isc").

The next step is to define a regular DHCPv4 option with our desired code and specify that it should include options from the new option space:

```
"Dhcp4": {
  "option-def": [
    ...,
    {
      "name": "container",
      "code": 222,
      "space": "dhcp4",
      "type": "empty",
      "array": false,
      "record-types": "",
      "encapsulate": "isc"
    }
  ],
  ...
}
```

The name of the option space in which the sub-options are defined is set in the **encapsulate** field. The **type** field is set to **empty**, to indicate that this option does not carry any data other than sub-options.

Finally, we can set values for the new options:



```
"Dhcp4": {
  "option-data": [
    {
      "name": "subopt1",
      "code": 1,
      "space": "isc",
      "data": "192.0.2.3"
    },
    {
      "name": "subopt2",
      "code": 2,
      "space": "isc",
      "data": "Hello world"
    },
    {
      "name": "container",
      "code": 222,
      "space": "dhcp4"
    }
  ],
  ...
}
```

Note that it is possible to create an option which carries some data in addition to the sub-options defined in the encapsulated option space. For example, if the "container" option from the previous example were required to carry a uint16 value as well as the sub-options, the **type** value would have to be set to "uint16" in the option definition. (Such an option would then have the following data structure: DHCP header, uint16 value, sub-options.) The value specified with the **data** parameter — which should be a valid integer enclosed in quotes, e.g. "123" — would then be assigned to the uint16 field in the "container" option.

8.2.15 Unspecified Parameters for DHCPv4 Option Configuration

In many cases it is not required to specify all parameters for an option configuration and the default values can be used. However, it is important to understand the implications of not specifying some of them, as it may result in configuration errors. The list below explains the behavior of the server when a particular parameter is not explicitly specified:

- **name** - the server requires an option name or option code to identify an option. If this parameter is unspecified, the option code must be specified.
- **code** - the server requires an option name or option code to identify an option. This parameter may be left unspecified if the **name** parameter is specified. However, this also requires that the particular option has its definition (it is either a standard option or an administrator created a definition for the option using an 'option-def' structure), as the option definition associates an option with a particular name. It is possible to configure an option for which there is no definition (unspecified option format). Configuration of such options requires the use of the option code.
- **space** - if the option space is unspecified it will default to 'dhcp4', which is an option space holding the standard DHCPv4 options.
- **data** - if the option data is unspecified it defaults to an empty value. The empty value is mostly used for the options which have no payload (boolean options), but it is legal to specify empty values for some options which carry variable-length data and which the specification allows to have a length of 0. For such options, the data parameter may be omitted in the configuration.
- **csv-format** - if this value is not specified, the server will assume that the option data is specified as a list of comma-separated values to be assigned to individual fields of the DHCP option.

8.2.16 Stateless Configuration of DHCPv4 Clients

The DHCPv4 server supports the stateless client configuration whereby the client has an IP address configured (e.g. using manual configuration) and only contacts the server to obtain other configuration parameters, e.g. addresses of DNS servers. In order to



obtain the stateless configuration parameters, the client sends the DHCPINFORM message to the server with the "ciaddr" set to the address that the client is currently using. The server unicasts the DHCPACK message to the client that includes the stateless configuration ("yiaddr" not set).

The server will respond to the DHCPINFORM when the client is associated with a subnet defined in the server's configuration. An example subnet configuration will look like this:

```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24"
      "option-data": [ {
        "name": "domain-name-servers",
        "code": 6,
        "data": "192.0.2.200,192.0.2.201",
        "csv-format": true,
        "space": "dhcp4"
      } ]
    } ]
  ]
}
```

This subnet specifies the single option which will be included in the DHCPACK message to the client in response to DHCPINFORM. Note that the subnet definition does not require the address pool configuration if it will be used solely for the stateless configuration.

This server will associate the subnet with the client if one of the following conditions is met:

- The DHCPINFORM is relayed and the giaddr matches the configured subnet.
- The DHCPINFORM is unicast from the client and the ciaddr matches the configured subnet.
- The DHCPINFORM is unicast from the client and the ciaddr is not set, but the source address of the IP packet matches the configured subnet.
- The DHCPINFORM is not relayed and the IP address on the interface on which the message is received matches the configured subnet.

8.2.17 Client Classification in DHCPv4

The DHCPv4 server includes support for client classification. For a deeper discussion of the classification process see Chapter 14.

In certain cases it is useful to configure the server to differentiate between DHCP client types and treat them accordingly. Client classification can be used to modify the behavior of almost any part of the DHCP message processing. Kea currently offers client classification via: private options and option 43 deferred unpacking; subnet selection; pool selection; assignment of different options; and, for cable modems, specific options for use with the TFTP server address and the boot file field.

Kea can be instructed to limit access to given subnets based on class information. This is particularly useful for cases where two types of devices share the same link and are expected to be served from two different subnets. The primary use case for such a scenario is cable networks, where there are two classes of devices: the cable modem itself, which should be handed a lease from subnet A; and all other devices behind the modem, which should get a lease from subnet B. That segregation is essential to prevent overly curious users from playing with their cable modems. For details on how to set up class restrictions on subnets, see Section 14.6.

When subnets belong to a shared network, the classification applies to subnet selection but not to pools, e.g. a pool in a subnet limited to a particular class can still be used by clients which do not belong to the class, if the pool they are expected to use is exhausted. So the limit on access based on class information is also available at the pool level; see Section 14.7, within a subnet. This is useful when segregating clients belonging to the same subnet into different address ranges.

In a similar way, a pool can be constrained to serve only known clients, i.e. clients which have a reservation, using the built-in "KNOWN" or "UNKNOWN" classes. One can assign addresses to registered clients without giving a different address per



reservation, for instance when there are not enough available addresses. The determination whether there is a reservation for a given client is made after a subnet is selected, so it is not possible to use KNOWN/UNKNOWN classes to select a shared network or a subnet.

The process of classification is conducted in five steps. The first step is to assess an incoming packet and assign it to zero or more classes. The second step is to choose a subnet, possibly based on the class information. When the incoming packet is in the special class, "DROP", it is dropped and a debug message logged. The next step is to evaluate class expressions depending on the built-in "KNOWN"/"UNKNOWN" classes after host reservation lookup, using them for pool selection and assigning classes from host reservations. The list of required classes is then built and each class of the list has its expression evaluated; when it returns "true" the packet is added as a member of the class. The last step is to assign options, again possibly based on the class information. More complete and detailed information is available in Chapter 14.

There are two main methods of classification. The first is automatic and relies on examining the values in the vendor class options or the existence of a host reservation. Information from these options is extracted, and a class name is constructed from it and added to the class list for the packet. The second specifies an expression that is evaluated for each packet. If the result is "true", the packet is a member of the class.

Note

Care should be taken with client classification, as it is easy for clients that do not meet class criteria to be denied all service.

8.2.17.1 Setting Fixed Fields in Classification

It is possible to specify that clients belonging to a particular class should receive packets with specific values in certain fixed fields. In particular, three fixed fields are supported: **next-server** (conveys an IPv4 address, which is set in the siaddr field), **server-hostname** (conveys a server hostname, can be up to 64 bytes long, and is sent in the sname field) and **boot-file-name** (conveys the configuration file, can be up to 128 bytes long, and is sent using the file field).

Obviously, there are many ways to assign clients to specific classes, but for the PXE clients the client architecture type option (code 93) seems to be particularly suited to make the distinction. The following example checks if the client identifies itself as a PXE device with architecture EFI x86-64, and sets several fields if it does. See [Section 2.1 of RFC 4578](#)) or the documentation of your client for specific values.

```
"Dhcp4": {
  "client-classes": [
    {
      "name": "ipxe_efi_x64",
      "test": "option[93].hex == 0x0009",
      "next-server": "192.0.2.254",
      "server-hostname": "hal9000",
      "boot-file-name": "/dev/null"
    },
    ...
  ],
  ...
}
```

If there are multiple classes defined and an incoming packet is matched to multiple classes, the class which is evaluated first is used.

Note

The classes are ordered as specified in the configuration.

8.2.17.2 Using Vendor Class Information in Classification

The server checks whether an incoming packet includes the vendor class identifier option (60). If it does, the content of that option is prepended with "VENDOR_CLASS_", and it is interpreted as a class. For example, modern cable modems will send this option with value "docsis3.0" and as a result the packet will belong to class "VENDOR_CLASS_docsis3.0".

**Note**

Certain special actions for clients in `VENDOR_CLASS_docsis3.0` can be achieved by defining `VENDOR_CLASS_docsis3.0` and setting its `next-server` and `boot-file-name` values appropriately.

This example shows a configuration using an automatically generated `"VENDOR_CLASS_"` class. The administrator of the network has decided that addresses from range 192.0.2.10 to 192.0.2.20 are going to be managed by the Dhcp4 server and only clients belonging to the `docsis3.0` client class are allowed to use that pool.

```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [ { "pool": "192.0.2.10 - 192.0.2.20" } ],
      "client-class": "VENDOR_CLASS_docsis3.0"
    }
  ],
  ...
}
```

8.2.17.3 Defining and Using Custom Classes

The following example shows how to configure a class using an expression and a subnet using that class. This configuration defines the class named `"Client_foo"`. It is comprised of all clients whose client ids (option 61) start with the string `"foo"`. Members of this class will be given addresses from 192.0.2.10 to 192.0.2.20 and the addresses of their DNS servers set to 192.0.2.1 and 192.0.2.2.

```
"Dhcp4": {
  "client-classes": [
    {
      "name": "Client_foo",
      "test": "substring(option[61].hex,0,3) == 'foo'",
      "option-data": [
        {
          "name": "domain-name-servers",
          "code": 6,
          "space": "dhcp4",
          "csv-format": true,
          "data": "192.0.2.1, 192.0.2.2"
        }
      ]
    }
  ],
  ...
},
"subnet4": [
  {
    "subnet": "192.0.2.0/24",
    "pools": [ { "pool": "192.0.2.10 - 192.0.2.20" } ],
    "client-class": "Client_foo"
  },
  ...
],
...
}
```

8.2.17.4 Required Classification

In some cases it is useful to limit the scope of a class to a shared-network, subnet, or pool. There are two parameters for this, which instruct the server to evaluate test expressions when required.



The first one is the per-class **only-if-required** flag which is false by default. When it is set to **true**, the test expression of the class is not evaluated at the reception of the incoming packet but later, and only if the class evaluation is required.

The second is **require-client-classes**, which takes a list of class names and is valid in shared-network, subnet, and pool scope. Classes in these lists are marked as required and evaluated after selection of this specific shared-network/subnet/pool and before output option processing.

In this example, a class is assigned to the incoming packet when the specified subnet is used:

```
"Dhcp4": {
  "client-classes": [
    {
      "name": "Client_foo",
      "test": "member('ALL')",
      "only-if-required": true
    },
    ...
  ],
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [ { "pool": "192.0.2.10 - 192.0.2.20" } ],
      "require-client-classes": [ "Client_foo" ],
      ...
    },
    ...
  ],
  ...
}
```

Required evaluation can be used to express complex dependencies, for example, subnet membership. It can also be used to reverse the precedence; if you set an option-data in a subnet, it takes precedence over an option-data in a class. When you move the option-data to a required class and require it in the subnet, a class evaluated earlier may take precedence.

Required evaluation is also available at shared-network and pool levels. The order in which required classes are considered is: shared-network, subnet, and pool, i.e. in the opposite order in which option-data is processed.

8.2.18 DDNS for DHCPv4

As mentioned earlier, kea-dhcp4 can be configured to generate requests to the DHCP-DDNS server (referred to here as "D2") to update DNS entries. These requests are known as Name Change Requests or NCRs. Each NCR contains the following information:

1. Whether it is a request to add (update) or remove DNS entries
2. Whether the change requests forward DNS updates (A records), reverse DNS updates (PTR records), or both
3. The Fully Qualified Domain Name (FQDN), lease address, and DHCID (information identifying the client associated with the FQDN)

The parameters for controlling the generation of NCRs for submission to D2 are contained in the **dhcp-ddns** section of the kea-dhcp4 server configuration. The mandatory parameters for the DHCP DDNS configuration are **enable-updates**, which is unconditionally required, and **qualifying-suffix**, which has no default value and is required when **enable-updates** is set to **true**. The two (disabled and enabled) minimal DHCP DDNS configurations are:

```
"Dhcp4": {
  "dhcp-ddns": {
    "enable-updates": false
  },
  ...
}
```



and for example:

```
"Dhcp4": {
  "dhcp-ddns": {
    "enable-updates": true,
    "qualifying-suffix": "example."
  },
  ...
}
```

The default values for the "dhcp-ddns" section are as follows:

- "server-ip": "127.0.0.1"
- "server-port": 53001
- "sender-ip": ""
- "sender-port": 0
- "max-queue-size": 1024
- "ncr-protocol": "UDP"
- "ncr-format": "JSON"
- "override-no-update": false
- "override-client-update": false
- "replace-client-name": "never"
- "generated-prefix": "myhost"
- "hostname-char-set": ""
- "hostname-char-replacement": ""

8.2.18.1 DHCP-DDNS Server Connectivity

For NCRs to reach the D2 server, kea-dhcp4 must be able to communicate with it. kea-dhcp4 uses the following configuration parameters to control this communication:

- **enable-updates** - determines whether kea-dhcp4 will generate NCRs. By default, this value is false, so DDNS updates are disabled. To enable DDNS updates set this value to true.
- **server-ip** - IP address on which D2 listens for requests. The default is the local loopback interface at address 127.0.0.1. You may specify either an IPv4 or IPv6 address.
- **server-port** - port on which D2 listens for requests. The default value is 53001.
- **sender-ip** - IP address which kea-dhcp4 uses to send requests to D2. The default value is blank, which instructs kea-dhcp4 to select a suitable address.
- **sender-port** - port which kea-dhcp4 uses to send requests to D2. The default value of 0 instructs kea-dhcp4 to select a suitable port.
- **max-queue-size** - maximum number of requests allowed to queue waiting to be sent to D2. This value guards against requests accumulating uncontrollably if they are being generated faster than they can be delivered. If the number of requests queued for transmission reaches this value, DDNS updating will be turned off until the queue backlog has been sufficiently reduced. The intent is to allow the kea-dhcp4 server to continue lease operations without running the risk that its memory usage grows without limit. The default value is 1024.



- **ncr-protocol** - socket protocol to use when sending requests to D2. Currently only UDP is supported.
- **ncr-format** - packet format to use when sending requests to D2. Currently only JSON format is supported.

By default, kea-dhcp-ddns is assumed to be running on the same machine as kea-dhcp4, and all of the default values mentioned above should be sufficient. If, however, D2 has been configured to listen on a different address or port, these values must be altered accordingly. For example, if D2 has been configured to listen on 192.168.1.10 port 900, the following configuration is required:

```
"Dhcp4": {
  "dhcp-ddns": {
    "server-ip": "192.168.1.10",
    "server-port": 900,
    ...
  },
  ...
}
```

8.2.18.2 When Does the kea-dhcp4 Server Generate a DDNS Request?

kea-dhcp4 follows the behavior prescribed for DHCP servers in [RFC 4702](#). It is important to keep in mind that kea-dhcp4 makes the initial decision of when and what to update and forwards that information to D2 in the form of NCRs. Carrying out the actual DNS updates and dealing with such things as conflict resolution are within the purview of D2 itself (Chapter 12). This section describes when kea-dhcp4 will generate NCRs and the configuration parameters that can be used to influence this decision. It assumes that the **enable-updates** parameter is true.

In general, kea-dhcp4 will generate DDNS update requests when:

1. A new lease is granted in response to a DHCPREQUEST
2. An existing lease is renewed but the FQDN associated with it has changed
3. An existing lease is released in response to a DHCPRELEASE

In the second case, lease renewal, two DDNS requests will be issued: one request to remove entries for the previous FQDN, and a second request to add entries for the new FQDN. In the last case, a lease release, a single DDNS request to remove its entries will be made.

The decisions involved when granting a new lease (the first case) are more complex. When a new lease is granted, kea-dhcp4 will generate a DDNS update request if the DHCPREQUEST contains either the FQDN option (code 81) or the Host Name option (code 12). If both are present, the server will use the FQDN option. By default, kea-dhcp4 will respect the FQDN N and S flags specified by the client as shown in the following table:

Client Flags:N-S	Client Intent	Server Response	Server Flags:N-S-O
0-0	Client wants to do forward updates, server should do reverse updates	Server generates reverse-only request	1-0-0
0-1	Server should do both forward and reverse updates	Server generates request to update both directions	0-1-0
1-0	Client wants no updates done	Server does not generate a request	1-0-0

Table 8.3: Default FQDN Flag Behavior

The first row in the table above represents "client delegation". Here the DHCP client states that it intends to do the forward DNS updates and the server should do the reverse updates. By default, kea-dhcp4 will honor the client's wishes and generate a DDNS request to the D2 server to update only reverse DNS data. The parameter **override-client-update** can be used to instruct the server to override client delegation requests. When this parameter is true, kea-dhcp4 will disregard requests for client delegation



and generate a DDNS request to update both forward and reverse DNS data. In this case, the N-S-O flags in the server's response to the client will be 0-1-1 respectively.

(Note that the flag combination N=1, S=1 is prohibited according to [RFC 4702](#). If such a combination is received from the client, the packet will be dropped by kea-dhcp4.)

To override client delegation, set the following values in the configuration file:

```
"Dhcp4": {
  "dhcp-ddns": {
    "override-client-update": true,
    ...
  },
  ...
}
```

The third row in the table above describes the case in which the client requests that no DNS updates be done. The parameter, **override-no-update**, can be used to instruct the server to disregard the client's wishes. When this parameter is true, kea-dhcp4 will generate DDNS update requests to kea-dhcp-ddns even if the client requests that no updates be done. The N-S-O flags in the server's response to the client will be 0-1-1.

To override client delegation, issue the following commands:

```
"Dhcp4": {
  "dhcp-ddns": {
    "override-no-update": true,
    ...
  },
  ...
}
```

kea-dhcp4 will always generate DDNS update requests if the client request only contains the Host Name option. In addition, it will include an FQDN option in the response to the client with the FQDN N-S-O flags set to 0-1-0 respectively. The domain name portion of the FQDN option will be the name submitted to D2 in the DDNS update request.

8.2.18.3 kea-dhcp4 Name Generation for DDNS Update Requests

Each NameChangeRequest must of course include the fully qualified domain name whose DNS entries are to be affected. kea-dhcp4 can be configured to supply a portion or all of that name, based upon what it receives from the client in the DHCPREQUEST.

The default rules for constructing the FQDN that will be used for DNS entries are:

1. If the DHCPREQUEST contains the client FQDN option, take the candidate name from there; otherwise, take it from the Host Name option.
2. If the candidate name is a partial (i.e. unqualified) name, then add a configurable suffix to the name and use the result as the FQDN.
3. If the candidate name provided is empty, generate an FQDN using a configurable prefix and suffix.
4. If the client provided neither option, then no DNS action will be taken.

These rules can be amended by setting the **replace-client-name** parameter, which provides the following modes of behavior:

- **never** - Use the name the client sent. If the client sent no name, do not generate one. This is the default mode.
- **always** - Replace the name the client sent. If the client sent no name, generate one for the client.
- **when-present** - Replace the name the client sent. If the client sent no name, do not generate one.
- **when-not-present** - Use the name the client sent. If the client sent no name, generate one for the client.

**Note**

Note that formerly, this parameter was a boolean and permitted only values of **true** and **false**. Boolean values have been deprecated and are no longer accepted. If you are currently using booleans, you must replace them with the desired mode name. A value of **true** maps to "**when-present**", while **false** maps to "**never**".

For example, to instruct kea-dhcp4 to always generate the FQDN for a client, set the parameter **replace-client-name** to **always** as follows:

```
"Dhcp4": {
  "dhcp-ddns": {
    "replace-client-name": "always",
    ...
  },
  ...
}
```

The prefix used in the generation of an FQDN is specified by the **generated-prefix** parameter. The default value is "myhost". To alter its value, simply set it to the desired string:

```
"Dhcp4": {
  "dhcp-ddns": {
    "generated-prefix": "another.host",
    ...
  },
  ...
}
```

The suffix used when generating an FQDN, or when qualifying a partial name, is specified by the **qualifying-suffix** parameter. This parameter has no default value, thus it is mandatory when DDNS updates are enabled. To set its value simply set it to the desired string:

```
"Dhcp4": {
  "dhcp-ddns": {
    "qualifying-suffix": "foo.example.org",
    ...
  },
  ...
}
```

When generating a name, kea-dhcp4 will construct the name in the format:

[generated-prefix]-[address-text].[qualifying-suffix].

where address-text is simply the lease IP address converted to a hyphenated string. For example, if the lease address is 172.16.1.10, the qualifying suffix "example.com", and the default value is used for **generated-prefix**, the generated FQDN would be:

myhost-172-16-1-10.example.com.

8.2.18.4 Sanitizing Client Host Name and FQDN Names

It may be that some of your DHCP clients provide values in the Host Name option (Option code 12) or FQDN option (Option code 81) that contain undesirable characters. It is possible to configure kea-dhcp4 to sanitize these values. The most typical use case is ensuring that only characters that are permitted by RFC 1035 be included: A-Z, a-z, 0-9, and '-'. This may be accomplished with the following two parameters:

- **hostname-char-set** - a regular expression describing the invalid character set. This can be any valid, regular expression using POSIX extended expression syntax. For example, "[^A-Za-z0-9-]" would replace any character other than the letters A through z, digits 0 through 9, and '-'. An empty string, the default value, disables sanitization.



- **hostname-char-replacement** - a string of zero or more characters with which to replace each invalid character in the host name. The default value is an empty string and will cause invalid characters to be OMITTED rather than replaced.

The following configuration will replace anything other than a letter, digit, hyphen, or dot with the letter 'x':

```
"Dhcp4": {
  "dhcp-ddns": {
    "hostname-char-set": "[^A-Za-z0-9.-]",
    "hostname-char-replacement": "x",
    ...
  },
  ...
}
```

Thus, a client-supplied value of "myhost-\$(123.org)" would become "myhost-xx123.org". Sanitizing is performed only on the portion of the name supplied by the client, and it is performed before applying a qualifying suffix (if one is defined and needed).

Note The following are some considerations to keep in mind:

Name sanitizing is meant to catch the more common cases of invalid characters through a relatively simple character-replacement scheme. It is difficult to devise a scheme that works well in all cases, for both Host Name and FQDN options. If you find you have clients that are using odd corner cases of character combinations that cannot be readily handled with this mechanism, you should consider writing a hook that can carry out sufficiently complex logic to address your needs.

If your clients include domain names in the Host Name option and you want these preserved, you will need to make sure that the dot, '.', is considered a valid character by the hostname-char-set expression, such as this: "[^A-Za-z0-9.-]". This will not affect dots in FQDN Option values. When scrubbing FQDNs, dots are treated as delimiters and used to separate the option value into individual domain labels that are scrubbed and then re-assembled.

If your clients are sending values that differ only by characters considered as invalid by your hostname-char-set, be aware that scrubbing them will yield identical values. In such cases, DDNS conflict rules will permit only one of them to register the name. Finally, given the latitude clients have in the values they send, it is virtually impossible to guarantee that a combination of these two parameters will always yield a name that is valid for use in DNS. For example, using an empty value for hostname-char-replacement could yield an empty domain label within a name, if that label consists only of invalid characters.

Note Since the 1.6.0 Kea release it is possible to specify hostname-char-set and/or hostname-char-replacement at the global scope. This allows to sanitize host names without requiring a dhcp-ddns entry. When a hostname-char parameter is defined at the global scope and in a dhcp-ddns entry the second (local) value is used.

8.2.19 Next Server (siaddr)

In some cases, clients want to obtain configuration from a TFTP server. Although there is a dedicated option for it, some devices may use the siaddr field in the DHCPv4 packet for that purpose. That specific field can be configured using the **next-server** directive. It is possible to define it in the global scope or for a given subnet only. If both are defined, the subnet value takes precedence. The value in subnet can be set to 0.0.0.0, which means that **next-server** should not be sent. It may also be set to an empty string, which means the same as if it were not defined at all, i.e. use the global value.

The **server-hostname** (which conveys a server hostname, can be up to 64 bytes long, and will be sent in the sname field) and **boot-file-name** (which conveys the configuration file, can be up to 128 bytes long, and will be sent using the file field) directives are handled the same way as **next-server**.

```
"Dhcp4": {
  "next-server": "192.0.2.123",
  "boot-file-name": "/dev/null",
  ...,
  "subnet4": [
    {
      "next-server": "192.0.2.234",
      "server-hostname": "some-name.example.org",

```



```
        "boot-file-name": "bootfile.efi",
        ...
    }
]
}
```

8.2.20 Echoing Client-ID (RFC 6842)

The original DHCPv4 specification ([RFC 2131](#)) states that the DHCPv4 server must not send back client-id options when responding to clients. However, in some cases that confused clients that did not have a MAC address or client-id; see [RFC 6842](#) for details. That behavior changed with the publication of [RFC 6842](#), which updated [RFC 2131](#). That update states that the server must send the client-id if the client sent it. That is Kea's default behavior. However, in some cases older devices that do not support [RFC 6842](#) may refuse to accept responses that include the client-id option. To enable backward compatibility, an optional configuration parameter has been introduced. To configure it, use the following configuration statement:

```
"Dhcp4": {
    "echo-client-id": false,
    ...
}
```

8.2.21 Using Client Identifier and Hardware Address

The DHCP server must be able to identify the client from which it receives the message and distinguish it from other clients. There are many reasons why this identification is required; the most important ones are:

- When the client contacts the server to allocate a new lease, the server must store the client identification information in the lease database as a search key.
- When the client is trying to renew or release the existing lease, the server must be able to find the existing lease entry in the database for this client, using the client identification information as a search key.
- Some configurations use static reservations for the IP addresses and other configuration information. The server's administrator uses client identification information to create these static assignments.
- In dual-stack networks there is often a need to correlate the lease information stored in DHCPv4 and DHCPv6 servers for a particular host. Using common identification information by the DHCPv4 and DHCPv6 clients allows the network administrator to achieve this correlation and better administer the network.

DHCPv4 uses two distinct identifiers which are placed by the client in the queries sent to the server and copied by the server to its responses to the client: "chaddr" and "client identifier". The former was introduced as a part of the BOOTP specification and it is also used by DHCP to carry the hardware address of the interface used to send the query to the server (MAC address for the Ethernet). The latter is carried in the Client-identifier option, introduced in [RFC 2132](#).

[RFC 2131](#) indicates that the server may use both of these identifiers to identify the client but the "client identifier", if present, takes precedence over "chaddr". One of the reasons for this is that "client identifier" is independent from the hardware used by the client to communicate with the server. For example, if the client obtained the lease using one network card and then the network card is moved to another host, the server will wrongly identify this host as the one which obtained the lease. Moreover, [RFC 4361](#) gives the recommendation to use a DUID (see [RFC 8415](#), the DHCPv6 specification) carried as "client identifier" when dual-stack networks are in use to provide consistent identification information for the client, regardless of the protocol type it is using. Kea adheres to these specifications, and the "client identifier" by default takes precedence over the value carried in the "chaddr" field when the server searches, creates, updates, or removes the client's lease.

When the server receives a DHCPDISCOVER or DHCPREQUEST message from the client, it will try to find out if the client already has a lease in the database and will hand out that lease rather than allocate a new one. Each lease in the lease database is associated with the "client identifier" and/or "chaddr". The server will first use the "client identifier" (if present) to search the lease. If the lease is found, the server will treat this lease as belonging to the client even if the current "chaddr" and the "chaddr" associated with the lease do not match. This facilitates the scenario when the network card on the client system has been replaced



and thus the new MAC address appears in the messages sent by the DHCP client. If the server fails to find the lease using the "client identifier", it will perform another lookup using the "chaddr". If this lookup returns no result, the client is considered as not having a lease and the new lease will be created.

A common problem reported by network operators is that poor client implementations do not use stable client identifiers, instead generating a new "client identifier" each time the client connects to the network. Another well-known case is when the client changes its "client identifier" during the multi-stage boot process (PXE). In such cases, the MAC address of the client's interface remains stable, and using the "chaddr" field to identify the client guarantees that the particular system is considered to be the same client, even though its "client identifier" changes.

To address this problem, Kea includes a configuration option which enables client identification using "chaddr" only by instructing the server to disregard the server to "ignore" the "client identifier" during lease lookups and allocations for a particular subnet. Consider the following simplified server configuration:

```
"Dhcp4": {
  ...
  "match-client-id": true,
  ...
  "subnet4": [
    {
      "subnet": "192.0.10.0/24",
      "pools": [ { "pool": "192.0.2.23-192.0.2.87" } ],
      "match-client-id": false
    },
    {
      "subnet": "10.0.0.0/8",
      "pools": [ { "pool": "10.0.0.23-10.0.2.99" } ],
    }
  ]
}
```

The **match-client-id** is a boolean value which controls this behavior. The default value of **true** indicates that the server will use the "client identifier" for lease lookups and "chaddr" if the first lookup returns no results. The **false** means that the server will only use the "chaddr" to search for client's lease. Whether the DHCPID for DNS updates is generated from the "client identifier" or "chaddr" is controlled through the same parameter.

The **match-client-id** parameter may appear both in the global configuration scope and/or under any subnet declaration. In the example shown above, the effective value of the **match-client-id** will be **false** for the subnet 192.0.10.0/24, because the subnet-specific setting of the parameter overrides the global value of the parameter. The effective value of the **match-client-id** for the subnet 10.0.0.0/8 will be set to **true** because the subnet declaration lacks this parameter and the global setting is by default used for this subnet. In fact, the global entry for this parameter could be omitted in this case, because **true** is the default value.

It is important to explain what happens when the client obtains its lease for one setting of the **match-client-id** and then renews when the setting has been changed. First, consider the case when the client obtains the lease when the **match-client-id** is set to **true**. The server will store the lease information, including "client identifier" (if supplied) and "chaddr", in the lease database. When the setting is changed and the client renews the lease, the server will determine that it should use the "chaddr" to search for the existing lease. If the client hasn't changed its MAC address, the server should successfully find the existing lease. The "client identifier" associated with the returned lease is ignored and the client is allowed to use this lease. When the lease is renewed only the "chaddr" is recorded for this lease, according to the new server setting.

In the second case the client has the lease with only a "chaddr" value recorded. When the **match-client-id** setting is changed to **true**, the server will first try to use the "client identifier" to find the existing client's lease. This will return no results because the "client identifier" was not recorded for this lease. The server will then use the "chaddr" and the lease will be found. If the lease appears to have no "client identifier" recorded, the server will assume that this lease belongs to the client and that it was created with the previous setting of the **match-client-id**. However, if the lease contains a "client identifier" which is different from the "client identifier" used by the client, the lease will be assumed to belong to another client and the new lease will be allocated.

8.2.22 Authoritative DHCPv4 Server Behavior

The original DHCPv4 specification ([RFC 2131](#)) states that if a client requests an address in the INIT-REBOOT state, of which the server has no knowledge, the server must remain silent, except if the server knows that the client has requested an IP address



from the wrong network. By default, Kea follows the behavior of the ISC `dhcpd` instead of the specification and also remains silent, if the client requests an IP address from the wrong network, because configuration information about a given network segment is not known to be correct. Kea only rejects a client's DHCPREQUEST with a DHCPNAK message if it already has a lease for the client, but with a different IP address. Administrators can override this behavior through the boolean **authoritative** (**false** by default) setting.

In authoritative mode, **authoritative** set to **true**, Kea always rejects INIT-REBOOT requests from unknown clients with DHCPNAK messages. The **authoritative** setting can be specified in global, shared-network, and subnet configuration scope and is automatically inherited from the parent scope, if not specified. All subnets in a shared-network must have the same **authoritative** setting.

8.2.23 DHCPv4-over-DHCPv6: DHCPv4 Side

The support of DHCPv4-over-DHCPv6 transport is described in [RFC 7341](#) and is implemented using cooperating DHCPv4 and DHCPv6 servers. This section is about the configuration of the DHCPv4 side (the DHCPv6 side is described in [Section 9.2.24](#)).

Note

DHCPv4-over-DHCPv6 support is experimental and the details of the inter-process communication may change; both the DHCPv4 and DHCPv6 sides should be running the same version of Kea. For instance, the support of port relay (RFC 8357) introduced an incompatible change.

The **dhcp4o6-port** global parameter specifies the first of the two consecutive ports of the UDP sockets used for the communication between the DHCPv6 and DHCPv4 servers (the DHCPv4 server is bound to `::1` on **port** + 1 and connected to `::1` on **port**).

With DHCPv4-over-DHCPv6, the DHCPv4 server does not have access to several of the identifiers it would normally use to select a subnet. To address this issue, three new configuration entries have been added; the presence of any of these allows the subnet to be used with DHCPv4-over-DHCPv6. These entries are:

- **4o6-subnet**: Takes a prefix (i.e., an IPv6 address followed by a slash and a prefix length) which is matched against the source address.
- **4o6-interface-id**: Takes a relay interface ID option value.
- **4o6-interface**: Takes an interface name which is matched against the incoming interface name.

The following configuration was used during some tests:

```
{
# DHCPv4 conf
"Dhcp4": {
  "interfaces-config": {
    "interfaces": [ "eno33554984" ]
  },
  "lease-database": {
    "type": "memfile",
    "name": "leases4"
  },
  "valid-lifetime": 4000,
  "subnet4": [ {
    "subnet": "10.10.10.0/24",
    "4o6-interface": "eno33554984",
    "4o6-subnet": "2001:db8:1:1::/64",
    "pools": [ { "pool": "10.10.10.100 - 10.10.10.199" } ]
  } ]
}
```



```
    } ],  
  
    "dhcp4o6-port": 6767,  
  
    "loggers": [ {  
        "name": "kea-dhcp4",  
        "output_options": [ {  
            "output": "/tmp/kea-dhcp4.log"  
        } ],  
        "severity": "DEBUG",  
        "debuglevel": 0  
    } ]  
}  
  
}
```

8.2.24 Sanity Checks in DHCPv4

An important aspect of a well-running DHCP system is an assurance that the data remains consistent. However, in some cases it may be convenient to tolerate certain inconsistent data. For example, a network administrator that temporarily removed a subnet from a configuration wouldn't want all the leases associated with it to disappear from the lease database. Kea has a mechanism to better control sanity checks such as this.

Kea supports a configuration scope called **sanity-checks**. It currently allows only a single parameter called **lease-checks**. It governs the verification that is done when a new lease is loaded from a lease file. With the sanity-checks mechanism, it is possible to tell Kea to try to correct inconsistent data.

Every subnet has a subnet-id value; this is how Kea internally identifies subnets. Each lease has a subnet-id parameter as well, which identifies which subnet it belongs to. However, if the configuration has changed, it is possible that a lease could exist with a subnet-id, but without any subnet that matches it. Also, it may be possible that the subnet's configuration has changed and the subnet-id now belongs to a subnet that does not match the lease. Kea's corrective algorithm first checks to see if there is a subnet with the subnet-id specified by the lease. If there is, it verifies whether the lease belongs to that subnet. If not, depending on the lease-checks setting, the lease is discarded, a warning is displayed, or a new subnet is selected for the lease that matches it topologically.

There are five levels which are supported:

- **none** - do no special checks; accept the lease as is.
- **warn** - if problems are detected, a warning will be displayed, but the lease data will be accepted anyway. This is the default value. If not explicitly configured to some other value, this level will be used.
- **fix** - If a data inconsistency is discovered, Kea will try to correct it. If the correction is not successful, the incorrect data will be inserted anyway.
- **fix-del** - If a data inconsistency is discovered, Kea will try to correct it. If the correction is not successful, the lease will be rejected. This setting ensures the data's correctness, but some incorrect data may be lost. Use with care.
- **del** - This is the strictest mode. If any inconsistency is detected, the lease is rejected. Use with care.

This feature is currently implemented for the memfile backend.

An example configuration that sets this parameter looks as follows:

```
"Dhcp4": {  
    "sanity-checks": {  
        "lease-checks": "fix-del"  
    },  
    ...  
}
```



8.3 Host Reservation in DHCPv4

There are many cases where it is useful to provide a configuration on a per-host basis. The most obvious one is to reserve a specific, static address for exclusive use by a given client (host); the returning client will receive the same address from the server every time, and other clients will generally not receive that address. Note that there may be cases when a new reservation has been made for a client for an address currently in use by another client. We call this situation a "conflict." These conflicts get resolved automatically over time as described in subsequent sections. Once the conflict is resolved, the client will keep receiving the reserved configuration when it renews.

Another example when host reservations are applicable is when a host has specific requirements, e.g. a printer that needs additional DHCP options. Yet another possible use case is to define unique names for hosts.

Host reservations are defined as parameters for each subnet. Each host must be identified by an identifier, for example the hardware/MAC address. There is an optional **reservations** array in the **subnet4** structure. Each element in that array is a structure that holds information about reservations for a single host. In particular, the structure must have an identifier that uniquely identifies a host. In the DHCPv4 context, the identifier is usually a hardware or MAC address. In most cases an IP address will be specified. It is also possible to specify a hostname, host specific options, or fields carried within DHCPv4 message such as `siaddr`, `sname`, or `file`.

The following example shows how to reserve addresses for specific hosts in a subnet:

```
"subnet4": [
  {
    "pools": [ { "pool": "192.0.2.1 - 192.0.2.200" } ],
    "subnet": "192.0.2.0/24",
    "interface": "eth0",
    "reservations": [
      {
        "hw-address": "1a:1b:1c:1d:1e:1f",
        "ip-address": "192.0.2.202"
      },
      {
        "duid": "0a:0b:0c:0d:0e:0f",
        "ip-address": "192.0.2.100",
        "hostname": "alice-laptop"
      },
      {
        "circuit-id": "'charter950'",
        "ip-address": "192.0.2.203"
      },
      {
        "client-id": "01:11:22:33:44:55:66",
        "ip-address": "192.0.2.204"
      }
    ]
  }
]
```

The first entry reserves the 192.0.2.202 address for the client that uses a MAC address of 1a:1b:1c:1d:1e:1f. The second entry reserves the address 192.0.2.100 and the hostname of alice-laptop for the client using a DUID 0a:0b:0c:0d:0e:0f. (Note that if you plan to do DNS updates, it is strongly recommended for the hostnames to be unique.) The third example reserves address 192.0.3.203 for a client whose request would be relayed by a relay agent that inserts a circuit-id option with the value 'charter950'. The fourth entry reserves address 192.0.2.204 for a client that uses a client identifier with value 01:11:22:33:44:55:66.

The above example is used for illustrational purposes only and in actual deployments it is recommended to use as few types as possible (preferably just one). See Section 8.3.8 for a detailed discussion of this point.

Making a reservation for a mobile host that may visit multiple subnets requires a separate host definition in each subnet it is expected to visit. It is not possible to define multiple host definitions with the same hardware address in a single subnet. Multiple host definitions with the same hardware address are valid if each is in a different subnet.

Adding host reservation incurs a performance penalty. In principle, when a server that does not support host reservation responds to a query, it needs to check whether there is a lease for a given address being considered for allocation or renewal. The server



that also supports host reservation has to perform additional checks: not only whether the address is currently used (i.e., if there is a lease for it), but also whether the address could be used by someone else (i.e., if there is a reservation for it). That additional check incurs extra overhead.

8.3.1 Address Reservation Types

In a typical scenario there is an IPv4 subnet defined, e.g. 192.0.2.0/24, with a certain part of it dedicated for dynamic allocation by the DHCPv4 server. That dynamic part is referred to as a dynamic pool or simply a pool. In principle, a host reservation can reserve any address that belongs to the subnet. The reservations that specify addresses that belong to configured pools are called "in-pool reservations." In contrast, those that do not belong to dynamic pools are called "out-of-pool reservations." There is no formal difference in the reservation syntax and both reservation types are handled uniformly.

Kea supports global host reservations. These are reservations that are specified at the global level within the configuration and that do not belong to any specific subnet. Kea will still match inbound client packets to a subnet as before, but when the subnet's reservation mode is set to "**global**", Kea will look for host reservations only among the global reservations defined. Typically, such reservations would be used to reserve hostnames for clients which may move from one subnet to another.

Note You can reserve any ip-address in a global reservation. Just keep in mind that Kea will not do any sanity checking on the address and for Kea 1.5.0, support for global reservations should be considered experimental.

8.3.2 Conflicts in DHCPv4 Reservations

As reservations and lease information are stored separately, conflicts may arise. Consider the following series of events: the server has configured the dynamic pool of addresses from the range of 192.0.2.10 to 192.0.2.20. Host A requests an address and gets 192.0.2.10. Now the system administrator decides to reserve address 192.0.2.10 for Host B. In general, reserving an address that is currently assigned to someone else is not recommended, but there are valid use cases where such an operation is warranted.

The server now has a conflict to resolve. If Host B boots up and requests an address, the server is not able to assign the reserved address 192.0.2.10. A naive approach would be to immediately remove the existing lease for Host A and create a new one for Host B. That would not solve the problem, though, because as soon as Host B gets the address, it will detect that the address is already in use by Host A and will send a DHCPDECLINE message. Therefore, in this situation, the server has to temporarily assign a different address from the dynamic pool (not matching what has been reserved) to Host B.

When Host A renews its address, the server will discover that the address being renewed is now reserved for another host - Host B. Therefore the server will inform Host A that it is no longer allowed to use it by sending a DHCPNAK message. The server will not remove the lease, though, as there's a small chance that the DHCPNAK may be lost if the network is lossy. If that happens, the client will not receive any responses, so it will retransmit its DHCPREQUEST packet. Once the DHCPNAK is received by Host A, it will revert to server discovery and will eventually get a different address. Besides allocating a new lease, the server will also remove the old one. As a result, address 192.0.2.10 will become free. When Host B tries to renew its temporarily assigned address, the server will detect that it has a valid lease, but will note that there is a reservation for a different address. The server will send DHCPNAK to inform Host B that its address is no longer usable, but will keep its lease (again, the DHCPNAK may be lost, so the server will keep it, until the client returns for a new address). Host B will revert to the server discovery phase and will eventually send a DHCPREQUEST message. This time the server will find that there is a reservation for that host and that the reserved address 192.0.2.10 is not used, so it will be granted. It will also remove the lease for the temporarily assigned address that Host B previously obtained.

This recovery will succeed, even if other hosts attempt to get the reserved address. If Host C requests the address 192.0.2.10 after the reservation is made, the server will either offer a different address (when responding to DHCPDISCOVER) or send DHCPNAK (when responding to DHCPREQUEST).

The recovery mechanism allows the server to fully recover from a case where reservations conflict with existing leases. This procedure takes time and will roughly take as long as the value set for renew-timer. The best way to avoid such recovery is not to define new reservations that conflict with existing leases. Another recommendation is to use out-of-pool reservations. If the reserved address does not belong to a pool, there is no way that other clients can get it.

**Note**

The conflict-resolution mechanism does not work for global reservations. As of Kea 1.5.0, it is generally recommended that you not use global reservations for addresses. If you choose to use them anyway, you must manually ensure that the reserved addresses are not in the dynamic pools.

8.3.3 Reserving a Hostname

When the reservation for a client includes the **hostname**, the server will return this hostname to the client in the Client FQDN or Hostname options. The server responds with the Client FQDN option only if the client has included Client FQDN option in its message to the server. The server will respond with the Hostname option if the client included Hostname option in its message to the server or when the client requested the Hostname option using the Parameter Request List option. The server will return the Hostname option even if it is not configured to perform DNS updates. The reserved hostname always takes precedence over the hostname supplied by the client or the autogenerated (from the IPv4 address) hostname.

The server qualifies the reserved hostname with the value of the **qualifying-suffix** parameter. For example, the following subnet configuration:

```
{
  "subnet4": [ {
    "subnet": "10.0.0.0/24",
    "pools": [ { "pool": "10.0.0.10-10.0.0.100" } ],
    "reservations": [
      {
        "hw-address": "aa:bb:cc:dd:ee:ff",
        "hostname": "alice-laptop"
      }
    ]
  } ],
  "dhcp-ddns": {
    "enable-updates": true,
    "qualifying-suffix": "example.isc.org."
  }
}
```

will result in assigning the "alice-laptop.example.isc.org." hostname to the client using the MAC address "aa:bb:cc:dd:ee:ff". If the **qualifying-suffix** is not specified, the default (empty) value will be used, and in this case the value specified as a **hostname** will be treated as a fully qualified name. Thus, by leaving the **qualifying-suffix** empty it is possible to qualify hostnames for different clients with different domain names:

```
{
  "subnet4": [ {
    "subnet": "10.0.0.0/24",
    "pools": [ { "pool": "10.0.0.10-10.0.0.100" } ],
    "reservations": [
      {
        "hw-address": "aa:bb:cc:dd:ee:ff",
        "hostname": "alice-laptop.isc.org."
      },
      {
        "hw-address": "12:34:56:78:99:AA",
        "hostname": "mark-desktop.example.org."
      }
    ]
  } ],
  "dhcp-ddns": {
    "enable-updates": true,
  }
}
```



8.3.4 Including Specific DHCPv4 Options in Reservations

Kea offers the ability to specify options on a per-host basis. These options follow the same rules as any other options. These can be standard options (see Section 8.2.10), custom options (see Section 8.2.11), or vendor-specific options (see Section 8.2.13). The following example demonstrates how standard options can be defined.

```
{
  "subnet4": [ {
    "reservations": [
      {
        "hw-address": "aa:bb:cc:dd:ee:ff",
        "ip-address": "192.0.2.1",
        "option-data": [
          {
            "name": "cookie-servers",
            "data": "10.1.1.202,10.1.1.203"
          },
          {
            "name": "log-servers",
            "data": "10.1.1.200,10.1.1.201"
          }
        ]
      }
    ]
  } ]
}
```

Vendor-specific options can be reserved in a similar manner:

```
{
  "subnet4": [ {
    "reservations": [
      {
        "hw-address": "aa:bb:cc:dd:ee:ff",
        "ip-address": "10.0.0.7",
        "option-data": [
          {
            "name": "vivso-suboptions",
            "data": "4491"
          },
          {
            "name": "tftp-servers",
            "space": "vendor-4491",
            "data": "10.1.1.202,10.1.1.203"
          }
        ]
      }
    ]
  } ]
}
```

Options defined at host level have the highest priority. In other words, if there are options defined with the same type on global, subnet, class, and host level, the host-specific values will be used.

8.3.5 Reserving Next Server, Server Hostname, and Boot File Name

BOOTP/DHCPv4 messages include "siaddr", "sname", and "file" fields. Even though DHCPv4 includes corresponding options, such as option 66 and option 67, some clients may not support these options. For this reason, server administrators often use the "siaddr", "sname", and "file" fields instead.

With Kea, it is possible to make static reservations for these DHCPv4 message fields:

```
{
  "subnet4": [ {
    "reservations": [
```



```
{
  "hw-address": "aa:bb:cc:dd:ee:ff",
  "next-server": "10.1.1.2",
  "server-hostname": "server-hostname.example.org",
  "boot-file-name": "/tmp/bootfile.efi"
} ]
} ]
}
```

Note that those parameters can be specified in combination with other parameters for a reservation, e.g. a reserved IPv4 address. These parameters are optional, i.e. a subset of them can be specified, or all of them can be omitted.

8.3.6 Reserving Client Classes in DHCPv4

Section 14.3 explains how to configure the server to assign classes to a client, based on the content of the options that this client sends to the server. Host reservations mechanisms also allow for the static assignment of classes to clients. The definitions of these classes are placed in the Kea configuration. The following configuration snippet shows how to specify that a client belongs to classes **reserved-class1** and **reserved-class2**. Those classes are associated with specific options being sent to the clients which belong to them.

```
{
  "client-classes": [
    {
      "name": "reserved-class1",
      "option-data": [
        {
          "name": "routers",
          "data": "10.0.0.200"
        }
      ]
    },
    {
      "name": "reserved-class2",
      "option-data": [
        {
          "name": "domain-name-servers",
          "data": "10.0.0.201"
        }
      ]
    }
  ],
  "subnet4": [ {
    "subnet": "10.0.0.0/24",
    "pools": [ { "pool": "10.0.0.10-10.0.0.100" } ],
    "reservations": [
      {
        "hw-address": "aa:bb:cc:dd:ee:ff",

        "client-classes": [ "reserved-class1", "reserved-class2" ]

      }
    ]
  } ]
}
```

Static class assignments, as shown above, can be used in conjunction with classification, using expressions. The "KNOWN" or "UNKNOWN" builtin class is added to the packet and any class depending on it (directly or indirectly) and not only-if-required is evaluated.

**Note**

If you want to force the evaluation of a class expression after the host reservation lookup, for instance because of a dependency on "reserved-class1" from the previous example, you should add a "member('KNOWN')" statement in the expression.

8.3.7 Storing Host Reservations in MySQL, PostgreSQL, or Cassandra

It is possible to store host reservations in MySQL, PostgreSQL, or Cassandra. See Section 9.2.3 for information on how to configure Kea to use reservations stored in MySQL, PostgreSQL, or Cassandra. Kea provides a dedicated hook for managing reservations in a database; section Section 15.4.4 provides detailed information. The Kea wiki <https://gitlab.isc.org/isc-projects/kea/wikis/designs/commands#23-host-reservations-hr-management> provides some examples of how to conduct common host reservation operations.

Note

In Kea, the maximum length of an option specified per-host is arbitrarily set to 4096 bytes.

8.3.8 Fine-Tuning DHCPv4 Host Reservation

The host reservation capability introduces additional restrictions for the allocation engine (the component of Kea that selects an address for a client) during lease selection and renewal. In particular, three major checks are necessary. First, when selecting a new lease, it is not sufficient for a candidate lease to simply not be in use by another DHCP client; it also must not be reserved for another client. Second, when renewing a lease, an additional check must be performed to see whether the address being renewed is reserved for another client. Finally, when a host renews an address, the server must check whether there is a reservation for this host, so the existing (dynamically allocated) address should be revoked and the reserved one be used instead.

Some of those checks may be unnecessary in certain deployments and not performing them may improve performance. The Kea server provides the **reservation-mode** configuration parameter to select the types of reservations allowed for a particular subnet. Each reservation type has different constraints for the checks to be performed by the server when allocating or renewing a lease for the client. Allowed values are:

- **all** - enables both in-pool and out-of-pool host reservation types. This is the default value. This setting is the safest and the most flexible. As all checks are conducted, it is also the slowest. This does not check against global reservations.
- **out-of-pool** - allows only out-of-pool host reservations. With this setting in place, the server may assume that all host reservations are for addresses that do not belong to the dynamic pool. Therefore, it can skip the reservation checks when dealing with in-pool addresses, thus improving performance. Do not use this mode if any of your reservations use in-pool addresses. Caution is advised when using this setting; Kea does not sanity-check the reservations against **reservation-mode** and misconfiguration may cause problems.
- **global** - allows only global host reservations. With this setting in place, the server searches for reservations for a client only among the defined global reservations. If an address is specified, the server will skip the reservation checks done when dealing in other modes, thus improving performance. Caution is advised when using this setting; Kea does not sanity-check the reservations when **global** and misconfiguration may cause problems.
- **disabled** - host reservation support is disabled. As there are no reservations, the server will skip all checks. Any reservations defined will be completely ignored. As the checks are skipped, the server may operate faster in this mode.

The parameter can be specified at global, subnet, and shared-network levels.

An example configuration that disables reservation looks as follows:

```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "reservation-mode": "disabled",
```



```
    ...
  }
]
}
```

An example configuration using global reservations is shown below:

```
"Dhcp4": {

  "reservation-mode": "global",
  "reservations": [
    {
      "hw-address": "01:bb:cc:dd:ee:ff",
      "hostname": "host-one"
    },
    {
      "hw-address": "02:bb:cc:dd:ee:ff",
      "hostname": "host-two"
    }
  ],

  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      ...
    }
  ]
}
```

For more details regarding global reservations, see Section [8.3.9](#).

Another aspect of the host reservations is the different types of identifiers. Kea currently supports four types of identifiers: hw-address, duid, client-id, and circuit-id. This is beneficial from a usability perspective; however, there is one drawback. For each incoming packet, Kea has to extract each identifier type and then query the database to see if there is a reservation by this particular identifier. If nothing is found, the next identifier is extracted and the next query is issued. This process continues until either a reservation is found or all identifier types have been checked. Over time, with an increasing number of supported identifier types, Kea would become slower and slower.

To address this problem, a parameter called **host-reservation-identifiers** is available. It takes a list of identifier types as a parameter. Kea will check only those identifier types enumerated in host-reservation-identifiers. From a performance perspective, the number of identifier types should be kept to a minimum, ideally one. If your deployment uses several reservation types, please enumerate them from most- to least-frequently used, as this increases the chances of Kea finding the reservation using the fewest queries. An example of host reservation identifiers looks as follows:

```
"host-reservation-identifiers": [ "circuit-id", "hw-address", "duid", "client-id" ],
"subnet4": [
  {
    "subnet": "192.0.2.0/24",
    ...
  }
]
```

If not specified, the default value is:

```
"host-reservation-identifiers": [ "hw-address", "duid", "circuit-id", "client-id" ]
```

8.3.9 Global Reservations in DHCPv4

In some deployments, such as mobile, clients can roam within the network and certain parameters must be specified regardless of the client's current location. To facilitate such a need, a global reservation mechanism has been implemented. The idea behind



it is that regular host reservations are tied to specific subnets, by using a specific subnet-id. Kea can specify a global reservation that can be used in every subnet that has global reservations enabled.

This feature can be used to assign certain parameters, such as hostname or other dedicated, host-specific options. It can also be used to assign addresses. However, global reservations that assign addresses bypass the whole topology determination provided by DHCP logic implemented in Kea. It is very easy to misuse this feature and get a configuration that is inconsistent. To give a specific example, imagine a global reservation for address 192.0.2.100 and two subnets 192.0.2.0/24 and 192.0.5.0/24. If global reservations are used in both subnets and a device matching global host reservations visits part of the network that is serviced by 192.0.5.0/24, it will get an IP address 192.0.2.100, a subnet 192.0.5.0 and a default router 192.0.5.1. Obviously, such a configuration is unusable, as the client won't be able to reach its default gateway.

To use global host reservations, a configuration similar to the following can be used:

```
"Dhcp4:" {
  // This specifies global reservations. They will apply to all subnets that
  // have global reservations enabled.

  "reservations": [
    {
      "hw-address": "aa:bb:cc:dd:ee:ff",
      "hostname": "hw-host-dynamic"
    },
    {
      "hw-address": "01:02:03:04:05:06",
      "hostname": "hw-host-fixed",

      // Use of IP address is global reservation is risky. If used outside of
      // matching subnet, such as 192.0.1.0/24, it will result in a broken
      // configuration being handled to the client.
      "ip-address": "192.0.1.77"
    },
    {
      "duid": "01:02:03:04:05",
      "hostname": "duid-host"
    },
    {
      "circuit-id": "'charter950'",
      "hostname": "circuit-id-host"
    },
    {
      "client-id": "01:11:22:33:44:55:66",
      "hostname": "client-id-host"
    }
  ],
  "valid-lifetime": 600,
  "subnet4": [ {
    "subnet": "10.0.0.0/24",
    "reservation-mode": "global",
    "pools": [ { "pool": "10.0.0.10-10.0.0.100" } ]
  } ]
}
```

When using database backends, the global host reservations are distinguished from regular reservations by using subnet-id value of zero.

8.4 Shared Networks in DHCPv4

DHCP servers use subnet information in two ways. First, it is used to determine the point of attachment, or simply put, where the client is connected to the network. Second, the subnet information is used to group information pertaining to a specific location in the network. This approach works well in general cases, but there are scenarios where the boundaries are blurred. Sometimes



it is useful to have more than one logical IP subnet deployed on the same physical link. The need to understand that two or more subnets are used on the same link requires additional logic in the DHCP server. This capability is called "shared networks" in Kea and ISC DHCP projects. It is sometimes also called "shared subnets." In Microsoft's nomenclature it is called "multinet."

There are many use cases where the feature is useful; this paragraph explains just a handful of the most common ones. The first and by far the most common use case is an existing network that has grown and is running out of available address space. Rather than migrating all devices to a new, larger subnet, it is easier to simply configure additional subnets on top of the existing one. Sometimes, due to address space fragmentation (e.g. only many disjointed /24s are available), this is the only choice. Also, configuring additional subnets has the advantage of not disrupting the operation of existing devices.

Another very frequent use case comes from cable networks. There are two types of devices in cable networks: cable modems and the end-user devices behind them. It is a common practice to use different subnets for cable modems to prevent users from tinkering with them. In this case, the distinction is based on the type of device, rather than address-space exhaustion.

A client connected to a shared network may be assigned an address from any of the pools defined within the subnets belonging to the shared network. Internally, the server selects one of the subnets belonging to a shared network and tries to allocate an address from this subnet. If the server is unable to allocate an address from the selected subnet (e.g., due to address pools exhaustion), it will use another subnet from the same shared network and try to allocate an address from this subnet, etc. Therefore, in the typical case, the server will allocate all addresses available in a given subnet before it starts allocating addresses from other subnets belonging to the same shared network. However, in certain situations the client can be allocated an address from the other subnets before the address pools in the first subnet get exhausted, e.g. when the client provides a hint that belongs to another subnet or the client has reservations in a subnet other than the default.

Note

Deployments should not assume that Kea waits until it has allocated all the addresses from the first subnet in a shared network before allocating addresses from other subnets.

In order to define a shared network an additional configuration scope is introduced:

```
{
  "Dhcp4": {
    "shared-networks": [
      {
        // Name of the shared network. It may be an arbitrary string
        // and it must be unique among all shared networks.
        "name": "my-secret-lair-level-1",

        // The subnet selector can be specified at the shared network level.
        // Subnets from this shared network will be selected for directly
        // connected clients sending requests to server's "eth0" interface.
        "interface": "eth0",

        // This starts a list of subnets in this shared network.
        // There are two subnets in this example.
        "subnet4": [
          {
            "subnet": "10.0.0.0/8",
            "pools": [ { "pool": "10.0.0.1 - 10.0.0.99" } ],
          },
          {
            "subnet": "192.0.2.0/24",
            "pools": [ { "pool": "192.0.2.100 - 192.0.2.199" } ]
          }
        ],
      },
    ], // end of shared-networks

    // It is likely that in your network you will have a mix of regular,
    // "plain" subnets and shared networks. It is perfectly valid to mix
    // them in the same configuration file.
    //
```



```
// This is regular subnet. It's not part of any shared-network.
"subnet4": [
  {
    "subnet": "192.0.3.0/24",
    "pools": [ { "pool": "192.0.3.1 - 192.0.3.200" } ],
    "interface": "eth1"
  }
]
} // end of Dhcp4
}
```

As you see in the example, it is possible to mix shared and regular ("plain") subnets. Each shared network must have a unique name. This is similar to the ID for subnets, but gives administrators more flexibility. It is used for logging, but also internally for identifying shared networks.

In principle it makes sense to define only shared networks that consist of two or more subnets. However, for testing purposes, an empty subnet or a network with just a single subnet is allowed. This is not a recommended practice in production networks, as the shared network logic requires additional processing and thus lowers the server's performance. To avoid unnecessary performance degradation, the shared subnets should only be defined when required by the deployment.

Shared networks provide an ability to specify many parameters in the shared network scope that will apply to all subnets within it. If necessary, you can specify a parameter in the shared network scope and then override its value in the subnet scope. For example:

```
"shared-networks": [
  {
    "name": "lab-network3",

    "interface": "eth0",

    // This applies to all subnets in this shared network, unless
    // values are overridden on subnet scope.
    "valid-lifetime": 600,

    // This option is made available to all subnets in this shared
    // network.
    "option-data": [ {
      "name": "log-servers",
      "data": "1.2.3.4"
    } ],

    "subnet4": [
      {
        "subnet": "10.0.0.0/8",
        "pools": [ { "pool": "10.0.0.1 - 10.0.0.99" } ],

        // This particular subnet uses different values.
        "valid-lifetime": 1200,
        "option-data": [
          {
            "name": "log-servers",
            "data": "10.0.0.254"
          },
          {
            "name": "routers",
            "data": "10.0.0.254"
          }
        ]
      },
      {
        "subnet": "192.0.2.0/24",
        "pools": [ { "pool": "192.0.2.100 - 192.0.2.199" } ],
```



```
// This subnet does not specify its own valid-lifetime value,  
// so it is inherited from shared network scope.  
"option-data": [  
  {  
    "name": "routers",  
    "data": "192.0.2.1"  
  } ]  
}  
]  
} ]
```

In this example, there is a log-servers option defined that is available to clients in both subnets in this shared network. Also, the valid lifetime is set to 10 minutes (600s). However, the first subnet overrides some of the values (valid lifetime is 20 minutes, different IP address for log-servers), but also adds its own option (router address). Assuming a client asking for router and log servers options is assigned a lease from this subnet, it will get a lease for 20 minutes and a log-servers and routers value of 10.0.0.254. If the same client is assigned to the second subnet, it will get a 10- minute lease, a log-servers value of 1.2.3.4, and routers set to 192.0.2.1.

8.4.1 Local and Relayed Traffic in Shared Networks

It is possible to specify an interface name in the shared network scope to tell the server that this specific shared network is reachable directly (not via relays) using a local network interface. It is sufficient to specify it once at the shared network level. As all subnets in a shared network are expected to be used on the same physical link, it is a configuration error to attempt to define a shared network using subnets that are reachable over different interfaces. It is possible to specify the interface parameter on each subnet, although its value must be the same for each subnet. Thus it is usually more convenient to specify it once at the shared network level.

```
"shared-networks": [  
  {  
    "name": "office-floor-2",  
  
    // This tells Kea that the whole shared networks is reachable over  
    // local interface. This applies to all subnets in this network.  
    "interface": "eth0",  
  
    "subnet4": [  
      {  
        "subnet": "10.0.0.0/8",  
        "pools": [ { "pool": "10.0.0.1 - 10.0.0.99" } ],  
        "interface": "eth0"  
      },  
      {  
        "subnet": "192.0.2.0/24",  
        "pools": [ { "pool": "192.0.2.100 - 192.0.2.199" } ]  
  
        // Specifying a different interface name is configuration  
        // error:  
        // "interface": "eth1"  
      }  
    ]  
  } ]
```

Somewhat similar to interface names, relay IP addresses can also be specified for the whole shared network. However, depending on your relay configuration, it may use different IP addresses depending on which subnet is being used. Thus there is no requirement to use the same IP relay address for each subnet. Here's an example:

```
"shared-networks": [  
  {  
    "name": "kakapo",
```



```
"relay": {
  "ip-addresses": [ "192.3.5.6" ]
},
"subnet4": [
  {
    "subnet": "192.0.2.0/26",
    "relay": {
      "ip-addresses": [ "192.1.1.1" ]
    },
    "pools": [ { "pool": "192.0.2.63 - 192.0.2.63" } ]
  },
  {
    "subnet": "10.0.0.0/24",
    "relay": {
      "ip-addresses": [ "192.2.2.2" ]
    },
    "pools": [ { "pool": "10.0.0.16 - 10.0.0.16" } ]
  }
]
}
```

In this particular case the relay IP address specified at the network level doesn't make much sense, as it is overridden in both subnets, but it was left there as an example of how one could be defined at the network level. Note that the relay agent IP address typically belongs to the subnet it relays packets from, but this is not a strict requirement. Kea accepts any value here as long as it is a valid IPv4 address.

8.4.2 Client Classification in Shared Networks

Sometimes it is desirable to segregate clients into specific subnets based on certain properties. This mechanism is called client classification and is described in Chapter 14. Client classification can be applied to subnets belonging to shared networks in the same way as it is used for subnets specified outside of shared networks. It is important to understand how the server selects subnets for clients when client classification is in use, to ensure that the desired subnet is selected for a given client type.

If a subnet is associated with a class, only the clients belonging to this class can use this subnet. If there are no classes specified for a subnet, any client connected to a given shared network can use this subnet. A common mistake is to assume that the subnet including a client class is preferred over subnets without client classes. Consider the following example:

```
{
  "client-classes": [
    {
      "name": "b-devices",
      "test": "option[93].hex == 0x0002"
    }
  ],
  "shared-networks": [
    {
      "name": "galah",
      "interface": "eth0",
      "subnet4": [
        {
          "subnet": "192.0.2.0/26",
          "pools": [ { "pool": "192.0.2.1 - 192.0.2.63" } ],
        },
        {
          "subnet": "10.0.0.0/24",
          "pools": [ { "pool": "10.0.0.2 - 10.0.0.250" } ],
          "client-class": "b-devices"
        }
      ]
    }
  ]
}
```



```
}  
]  
}
```

If the client belongs to the "b-devices" class (because it includes option 93 with a value of 0x0002), that doesn't guarantee that the subnet 10.0.0.0/24 will be used (or preferred) for this client. The server can use either of the two subnets because the subnet 192.0.2.0/26 is also allowed for this client. The client classification used in this case should be perceived as a way to restrict access to certain subnets, rather than a way to express subnet preference. For example, if the client doesn't belong to the "b-devices" class it may only use the subnet 192.0.2.0/26 and will never use the subnet 10.0.0.0/24.

A typical use case for client classification is in a cable network, where cable modems should use one subnet and other devices should use another subnet within the same shared network. In this case it is necessary to apply classification on all subnets. The following example defines two classes of devices, and the subnet selection is made based on option 93 values.

```
{  
  "client-classes": [  
    {  
      "name": "a-devices",  
      "test": "option[93].hex == 0x0001"  
    },  
    {  
      "name": "b-devices",  
      "test": "option[93].hex == 0x0002"  
    }  
  ],  
  "shared-networks": [  
    {  
      "name": "galah",  
      "interface": "eth0",  
      "subnet4": [  
        {  
          "subnet": "192.0.2.0/26",  
          "pools": [ { "pool": "192.0.2.1 - 192.0.2.63" } ],  
          "client-class": "a-devices"  
        },  
        {  
          "subnet": "10.0.0.0/24",  
          "pools": [ { "pool": "10.0.0.2 - 10.0.0.250" } ],  
          "client-class": "b-devices"  
        }  
      ]  
    }  
  ]  
}
```

In this example each class has its own restriction. Only clients that belong to class "a-devices" will be able to use subnet 192.0.2.0/26 and only clients belonging to "b-devices" will be able to use subnet 10.0.0.0/24. Care should be taken not to define too-restrictive classification rules, as clients that are unable to use any subnets will be refused service. However, this may be a desired outcome if one wishes to provide service only to clients with known properties (e.g. only VoIP phones allowed on a given link).

Note that it is possible to achieve an effect similar to the one presented in this section without the use of shared networks. If the subnets are placed in the global subnets scope, rather than in the shared network, the server will still use classification rules to pick the right subnet for a given class of devices. The major benefit of placing subnets within the shared network is that common parameters for the logically grouped subnets can be specified once, in the shared network scope, e.g. the "interface" or "relay" parameter. All subnets belonging to this shared network will inherit those parameters.

8.4.3 Host Reservations in Shared Networks

Subnets that are part of a shared network allow host reservations, similar to regular subnets:



```
{
  "shared-networks": [
    {
      "name": "frog",
      "interface": "eth0",
      "subnet4": [
        {
          "subnet": "192.0.2.0/26",
          "id": 100,
          "pools": [ { "pool": "192.0.2.1 - 192.0.2.63" } ],
          "reservations": [
            {
              "hw-address": "aa:bb:cc:dd:ee:ff",
              "ip-address": "192.0.2.28"
            }
          ]
        }
      ],
    },
    {
      "subnet": "10.0.0.0/24",
      "id": 101,
      "pools": [ { "pool": "10.0.0.1 - 10.0.0.254" } ],
      "reservations": [
        {
          "hw-address": "11:22:33:44:55:66",
          "ip-address": "10.0.0.29"
        }
      ]
    }
  ]
}
}
```

It is worth noting that Kea conducts additional checks when processing a packet if shared networks are defined. First, instead of simply checking whether there's a reservation for a given client in its initially selected subnet, Kea looks through all subnets in a shared network for a reservation. This is one of the reasons why defining a shared network may impact performance. If there is a reservation for a client in any subnet, that particular subnet will be picked for the client. Although it's technically not an error, it is considered a bad practice to define reservations for the same host in multiple subnets belonging to the same shared network.

While not strictly mandatory, it is strongly recommended to use explicit "id" values for subnets if you plan to use database storage for host reservations. If an ID is not specified, the values for it are autogenerated, i.e. it assigns increasing integer values starting from 1. Thus, the autogenerated IDs are not stable across configuration changes.

8.5 Server Identifier in DHCPv4

The DHCPv4 protocol uses a "server identifier" to allow clients to discriminate between several servers present on the same link; this value is an IPv4 address of the server. The server chooses the IPv4 address of the interface on which the message from the client (or relay) has been received. A single server instance will use multiple server identifiers if it is receiving queries on multiple interfaces.

It is possible to override the default server identifier values by specifying the "dhcp-server-identifier" option. This option is only supported at the global, shared network, and subnet levels. It must not be specified on the client class and host reservation levels.

The following example demonstrates how to override the server identifier for a subnet:

```
"subnet4": [
  {
    "subnet": "192.0.2.0/24",
    "option-data": [
```



```
    {
      "name": "dhcp-server-identifier",
      "data": "10.2.5.76"
    }
  ],
  ...
}
```

8.6 How the DHCPv4 Server Selects a Subnet for the Client

The DHCPv4 server differentiates between directly connected clients, clients trying to renew leases, and clients sending their messages through relays. For directly connected clients, the server will check the configuration for the interface on which the message has been received and, if the server configuration doesn't match any configured subnet, the message is discarded.

Assuming that the server's interface is configured with the IPv4 address 192.0.2.3, the server will only process messages received through this interface from a directly connected client if there is a subnet configured to which this IPv4 address belongs, e.g. 192.0.2.0/24. The server will use this subnet to assign an IPv4 address for the client.

The rule above does not apply when the client unicasts its message, i.e. is trying to renew its lease. Such a message is accepted through any interface. The renewing client sets `ciaddr` to the currently used IPv4 address, and the server uses this address to select the subnet for the client (in particular, to extend the lease using this address).

If the message is relayed it is accepted through any interface. The `giaddr` set by the relay agent is used to select the subnet for the client.

It is also possible to specify a relay IPv4 address for a given subnet. It can be used to match incoming packets into a subnet in uncommon configurations, e.g. shared networks. See Section 8.6.1 for details.

Note

The subnet selection mechanism described in this section is based on the assumption that client classification is not used. The classification mechanism alters the way in which a subnet is selected for the client, depending on the classes to which the client belongs.

8.6.1 Using a Specific Relay Agent for a Subnet

A relay must have an interface connected to the link on which the clients are being configured. Typically the relay has an IPv4 address configured on that interface, which belongs to the subnet from which the server will assign addresses. Normally, the server is able to use the IPv4 address inserted by the relay (in the `giaddr` field of the DHCPv4 packet) to select the appropriate subnet.

However, that is not always the case. In certain uncommon — but valid — deployments, the relay address may not match the subnet. This usually means that there is more than one subnet allocated for a given link. The two most common examples where this is the case are long-lasting network renumbering (where both old and new address space is still being used) and a cable network. In a cable network, both cable modems and the devices behind them are physically connected to the same link, yet they use distinct addressing. In such a case, the DHCPv4 server needs additional information (the IPv4 address of the relay) to properly select an appropriate subnet.

The following example assumes that there is a subnet 192.0.2.0/24 that is accessible via a relay that uses 10.0.0.1 as its IPv4 address. The server is able to select this subnet for any incoming packets that come from a relay that has an address in the 192.0.2.0/24 subnet. It also selects that subnet for a relay with address 10.0.0.1.

```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [ { "pool": "192.0.2.10 - 192.0.2.20" } ],
    }
  ]
}
```



```
    "relay": {
      "ip-addresses": [ "10.0.0.1" ]
    },
    ...
  }
],
...
}
```

If "relay" is specified, the "ip-addresses" parameter within it is mandatory.

Note

The current version of Kea uses the "ip-addresses" parameter, which supports specifying a list of addresses.

8.6.2 Segregating IPv4 Clients in a Cable Network

In certain cases, it is useful to mix relay address information, introduced in Section 8.6.1, with client classification, explained in Chapter 14. One specific example is in a cable network, where typically modems get addresses from a different subnet than all the devices connected behind them.

Let us assume that there is one CMTS (Cable Modem Termination System) with one CM MAC (a physical link that modems are connected to). We want the modems to get addresses from the 10.1.1.0/24 subnet, while everything connected behind the modems should get addresses from another subnet (192.0.2.0/24). The CMTS that acts as a relay uses address 10.1.1.1. The following configuration can serve that configuration:

```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "10.1.1.0/24",
      "pools": [ { "pool": "10.1.1.2 - 10.1.1.20" } ],
      "client-class" "docsis3.0",
      "relay": {
        "ip-addresses": [ "10.1.1.1" ]
      }
    },
    {
      "subnet": "192.0.2.0/24",
      "pools": [ { "pool": "192.0.2.10 - 192.0.2.20" } ],
      "relay": {
        "ip-addresses": [ "10.1.1.1" ]
      }
    }
  ],
  ...
}
```

8.7 Duplicate Addresses (DHCPDECLINE Support)

The DHCPv4 server is configured with a certain pool of addresses that it is expected to hand out to DHCPv4 clients. It is assumed that the server is authoritative and has complete jurisdiction over those addresses. However, for various reasons, such as misconfiguration or a faulty client implementation that retains its address beyond the valid lifetime, there may be devices connected that use those addresses without the server's approval or knowledge.

Such an unwelcome event can be detected by legitimate clients (using ARP or ICMP Echo Request mechanisms) and reported to the DHCPv4 server using a DHCPDECLINE message. The server will do a sanity check (to see whether the client declining an address really was supposed to use it), and then will conduct a clean-up operation. Any DNS entries related to that address will



be removed, the fact will be logged, and hooks will be triggered. After that is complete, the address will be marked as declined (which indicates that it is used by an unknown entity and thus not available for assignment) and a probation time will be set on it. Unless otherwise configured, the probation period lasts 24 hours; after that period, the server will recover the lease (i.e. put it back into the available state) and the address will be available for assignment again. It should be noted that if the underlying issue of a misconfigured device is not resolved, the duplicate-address scenario will repeat. If reconfigured correctly, this mechanism provides an opportunity to recover from such an event automatically, without any sysadmin intervention.

To configure the decline probation period to a value other than the default, the following syntax can be used:

```
"Dhcp4": {
  "decline-probation-period": 3600,
  "subnet4": [ ... ],
  ...
}
```

The parameter is expressed in seconds, so the example above will instruct the server to recycle declined leases after one hour.

There are several statistics and hook points associated with the Decline handling procedure. The `lease4_decline` hook is triggered after the incoming DHCPDECLINE message has been sanitized and the server is about to decline the lease. The `declined-addresses` statistic is increased after the hook returns (both global and subnet-specific variants). (See Section 8.8 and Chapter 15 for more details on DHCPv4 statistics and Kea hook points.)

Once the probation time elapses, the declined lease is recovered using the standard expired-lease reclamation procedure, with several additional steps. In particular, both `declined-addresses` statistics (global and subnet-specific) are decreased. At the same time, `reclaimed-declined-addresses` statistics (again in two variants, global and subnet-specific) are increased.

A note about statistics: The server does not decrease the `assigned-addresses` statistics when a DHCPDECLINE is received and processed successfully. While technically a declined address is no longer assigned, the primary usage of the `assigned-addresses` statistic is to monitor pool utilization. Most people would forget to include `declined-addresses` in the calculation, and simply use `assigned-addresses/total-addresses`. This would cause a bias towards under-representing pool utilization. As this has a potential for major issues, we decided not to decrease `assigned-addresses` immediately after receiving DHCPDECLINE, but to do it later when Kea recovers the address back to the available pool.

8.8 Statistics in the DHCPv4 Server

Note

This section describes DHCPv4-specific statistics. For a general overview and usage of statistics, see Chapter 16.

The DHCPv4 server supports the following statistics:

8.9 Management API for the DHCPv4 Server

The management API allows the issuing of specific management commands, such as statistics retrieval, reconfiguration, or shutdown. For more details, see Chapter 17. Currently, the only supported communication channel type is UNIX stream socket. By default there are no sockets open; to instruct Kea to open a socket, the following entry in the configuration file can be used:

```
"Dhcp4": {
  "control-socket": {
    "socket-type": "unix",
    "socket-name": "/path/to/the/unix/socket"
  },

  "subnet4": [
    ...
  ],
  ...
}
```



Statistic	Data Type	Description
pkt4-received	integer	Number of DHCPv4 packets received. This includes all packets: valid, bogus, corrupted, rejected, etc. This statistic is expected to grow rapidly.
pkt4-discover-received	integer	Number of DHCPDISCOVER packets received. This statistic is expected to grow; its increase means that clients that just booted started their configuration process and their initial packets reached your Kea server.
pkt4-offer-received	integer	Number of DHCPOFFER packets received. This statistic is expected to remain zero at all times, as DHCPOFFER packets are sent by the server and the server is never expected to receive them. A non-zero value indicates an error. One likely cause would be a misbehaving relay agent that incorrectly forwards DHCPOFFER messages towards the server, rather than back to the clients.
pkt4-request-received	integer	Number of DHCPREQUEST packets received. This statistic is expected to grow. Its increase means that clients that just booted received the server's response (DHCPOFFER) and accepted it, and are now requesting an address (DHCPREQUEST).
pkt4-ack-received	integer	Number of DHCPACK packets received. This statistic is expected to remain zero at all times, as DHCPACK packets are sent by the server and the server is never expected to receive them. A non-zero value indicates an error. One likely cause would be a misbehaving relay agent that incorrectly forwards DHCPACK messages towards the server, rather than back to the clients.
pkt4-nak-received	integer	Number of DHCPNAK packets received. This statistic is expected to remain zero at all times, as DHCPNAK packets are sent by the server and the server is never expected to receive them. A non-zero value indicates an error. One likely cause would be a misbehaving relay agent that incorrectly forwards DHCPNAK messages towards the server, rather than back to the clients.
pkt4-release-received	integer	Number of DHCPRELEASE packets received. This statistic is expected to grow. Its increase means that clients that had an address are shutting down or ceasing to use their addresses.
pkt4-decline-received	integer	Number of DHCPDECLINE packets received. This statistic is expected to remain close to zero. Its increase means that a client leased an address, but discovered that the address is currently used by an unknown device in your network.
		Number of DHCPINFORM packets



The length of the path specified by the **socket-name** parameter is restricted by the maximum length for the UNIX socket name on your operating system, i.e. the size of the **sun_path** field in the **sockaddr_un** structure, decreased by 1. This value varies on different operating systems between 91 and 107 characters. Typical values are 107 on Linux and 103 on FreeBSD.

Communication over the control channel is conducted using JSON structures. See the `Control Channel` section in the *Kea Developer's Guide* for more details.

The DHCPv4 server supports the following operational commands:

- build-report
- config-get
- config-reload
- config-set
- config-test
- config-write
- dhcp-disable
- dhcp-enable
- leases-reclaim
- list-commands
- shutdown
- version-get

as described in Section 17.3. In addition, it supports the following statistics-related commands:

- statistic-get
- statistic-reset
- statistic-remove
- statistic-get-all
- statistic-reset-all
- statistic-remove-all

as described in Section 16.3.

8.10 User Contexts in IPv4

Kea allows loading hook libraries that sometimes could benefit from additional parameters. If such a parameter is specific to the whole library, it is typically defined as a parameter for the hook library. However, sometimes there is a need to specify parameters that are different for each pool.

User contexts can store arbitrary data as long as it has valid JSON syntax and its top level element is a map (i.e. the data must be enclosed in curly brackets). However, some hook libraries may expect specific formatting; please consult the specific hook library documentation for details.

User contexts can be specified at global scope, shared network, subnet, pool, client class, option data, or definition level, and via host reservation. One other useful usage is the ability to store comments or descriptions.

Let's consider an imaginary case of devices that have color LED lights. Depending on their location, they should glow red, blue, or green. It would be easy to write a hook library that would send specific values as maybe a vendor option. However, the server has to have some way to specify that value for each pool. This need is addressed by user contexts. In essence, any user data can be specified in the user context as long as it is a valid JSON map. For example, the forementioned case of LED devices could be configured in the following way:



```
"Dhcp4": {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [ {
        "pool": "192.0.2.10 - 192.0.2.20",
        // This is pool specific user context
        "user-context": { "color": "red" }
      } ],

      // This is a subnet specific user context. You can put whatever type
      // of information you want as long as it is a valid JSON.
      "user-context": {
        "comment": "network on the second floor",
        "last-modified": "2017-09-04 13:32",
        "description": "you can put here anything you like",
        "phones": [ "x1234", "x2345" ],
        "devices-registered": 42,
        "billing": false
      }
    },
    ...
  ],
  ...
}
```

Kea does not use that information; it simply stores it and makes it available to the hook libraries. It is up to each hook library to extract that information and use it. The parser translates a "comment" entry into a user context with the entry, which allows a comment to be attached inside the configuration itself.

For more background information, see Section [15.5](#).

8.11 Supported DHCP Standards

The following standards are currently supported:

- *Dynamic Host Configuration Protocol*, [RFC 2131](#): Supported messages are DHCPDISCOVER (1), DHCPOFFER (2), DHCPREQUEST (3), DHCPRELEASE (7), DHCPINFORM (8), DHCPACK (5), and DHCPNAK(6).
- *DHCP Options and BOOTP Vendor Extensions*, [RFC 2132](#): Supported options are: PAD (0), END(255), Message Type(53), DHCP Server Identifier (54), Domain Name (15), DNS Servers (6), IP Address Lease Time (51), Subnet mask (1), and Routers (3).
- *DHCP Relay Agent Information Option*, [RFC 3046](#): Relay Agent Information option is supported.
- *Vendor-Identifying Vendor Options for Dynamic Host Configuration Protocol version 4*, [RFC 3925](#): Vendor-Identifying Vendor Class and Vendor-Identifying Vendor-Specific Information options are supported.
- *Client Identifier Option in DHCP Server Replies*, [RFC 6842](#): Server by default sends back client-id option. That capability may be disabled. See Section [8.2.20](#) for details.

8.12 DHCPv4 Server Limitations

These are the current limitations of the DHCPv4 server software. Most of them are reflections of the current stage of development and should be treated as “not implemented yet”, rather than actual limitations. However, some of them are implications of the design choices made. Those are clearly marked as such.



- BOOTP ([RFC 951](#)) is not supported. This is a design choice: BOOTP support is not planned.
- On Linux and BSD system families the DHCP messages are sent and received over the raw sockets (using LPF and BPF) and all packet headers (including data link layer, IP, and UDP headers) are created and parsed by Kea, rather than by the system kernel. Currently, Kea can only parse the data link layer headers with a format adhering to the IEEE 802.3 standard and assumes this data link layer header format for all interfaces. Hence, Kea will fail to work on interfaces which use different data link layer header formats (e.g. Infiniband).
- The DHCPv4 server does not verify that an assigned address is unused. According to [RFC 2131](#), the allocating server should verify that an address is not used by sending an ICMP echo request.

8.13 Kea DHCPv4 Server Examples

A collection of simple-to-use examples for the DHCPv4 component of Kea is available with the source files, located in the `doc/examples/kea4` directory.

8.14 Configuration Backend in DHCPv4

In the [Section 5.2](#) we have described the Configuration Backend feature, its applicability and limitations. This section focuses on the usage of the CB with the DHCPv4 server. It lists the supported parameters, describes limitations and gives examples of the DHCPv4 server configuration to take advantage of the CB. Please also refer to the sibling section [Section 9.19](#) for the DHCPv6 specific usage of the CB.

8.14.1 Supported Parameters

The ultimate goal for the CB is to serve as a central configuration repository for one or multiple Kea servers connected to the database. In the future it will be possible to store the most of the server's configuration in the database and reduce the configuration file to bare minimum, i.e. the only mandatory parameter will be the **config-control** which includes the necessary information to connect to the database. In the Kea 1.6.0 release, however, only the subset of the DHCPv4 server parameters can be stored in the database. All other parameters must be specified in the JSON configuration file, if required.

The following table lists DHCPv4 specific parameters supported by the Configuration Backend with an indication on which level of the hierarchy it is currently supported. The "n/a" is used in cases when the particular parameter is not applicable on the particular level of the hierarchy or in cases when the parameter is not supported by the server on this level of hierarchy. The "no" is used when the parameter is supported by the server on the particular level of hierarchy but is not configurable via the Configuration Backend.

All supported parameters can be configured via **cb_cmds** hooks library described in the [Section 15.4.8](#). The general rule is that the scalar global parameters are set using the **remote-global-parameter4-set**. The shared network specific parameters are set using the **remote-network4-set**. Finally, the subnet and pool level parameters are set using the **remote-subnet4-set**. Whenever there is an exception from this general rule, it is highlighted in the table. The non-scalar global parameters have dedicated commands, e.g. modifying the global DHCPv4 options (**option-data**) is performed using the **remote-option4-global-set**.

The [Section 5.2.4](#) explains the concept of shareable and non-shareable configuration elements and the limitations for sharing them between multiple servers. In the DHCP configuration (both DHCPv4 and DHCPv6) the shareable configuration elements are: subnets and shared networks. Thus, they can be explicitly associated with multiple server tags. The global parameters, option definitions and global options are non-shareable and they can be associated with only one server tag. This rule does not apply to the configuration elements associated with "all" servers. Any configuration element associated with "all" servers (using "all" keyword as a server tag) is used by all servers connecting to the configuration database.

8.14.2 Enabling Configuration Backend

Consider the following configuration snippet:



Parameter	Global	Shared Network	Subnet	Pool
4o6-interface	n/a	n/a	yes	n/a
4o6-interface-id	n/a	n/a	yes	n/a
4o6-subnet	n/a	n/a	yes	n/a
boot-file-name	yes	yes	yes	n/a
calculate-tee-times	yes	yes	yes	n/a
client-class	n/a	yes	yes	no
decline-probation-period	yes	n/a	n/a	n/a
dhcp4o6-port	yes	n/a	n/a	n/a
echo-client-id	yes	n/a	n/a	n/a
interface	n/a	yes	yes	n/a
match-client-id	yes	yes	yes	n/a
next-server	yes	yes	yes	n/a
option-data	yes (via remote-option4-global-set)	yes	yes	yes
option-def	yes (via remote-option-def4-set)	n/a	n/a	n/a
rebind-timer	yes	yes	yes	n/a
renew-timer	yes	yes	yes	n/a
server-hostname	yes	yes	yes	n/a
valid-lifetime	yes	yes	yes	n/a
relay	n/a	yes	yes	n/a
require-client-classes	no	yes	yes	no
reservation-mode	yes	yes	yes	n/a
t1-percent	yes	yes	yes	n/a
t2-percent	yes	yes	yes	n/a

Table 8.5: List of DHCPv4 Parameters Supported by the Configuration Backend



```
{
  "Dhcp4": {
    "config-control": {
      "config-databases": [
        {
          "type": "mysql",
          "name": "kea",
          "user": "kea",
          "password": "kea",
          "host": "192.0.2.1",
          "port": 3302
        }
      ],
      "config-fetch-wait-time": 20
    },
    "hooks-libraries": [
      {
        "library": "/usr/local/lib/kea/hooks/libdhcp_mysql_cb.so"
      },
      {
        "library": "/usr/local/lib/kea/hooks/libdhcp_cb_cmds.so"
      }
    ],
    ...
  }
}
```

The **config-control** contains two parameters. The **config-databases** is a list which contains one element comprising database type, location and the credentials to be used to connect to this database. Note that the parameters specified here correspond to the database specification for the lease database backend and hosts database backend. Currently only one database connection can be specified on the **config-databases** list. The server will connect to this database during the startup or reconfiguration, and will fetch the configuration available for this server from the database. This configuration is merged into the configuration read from the configuration file.

Note

Whenever there is a conflict between the parameters specified in the configuration file and the database, the parameters from the database take precedence. We strongly recommend to avoid duplicating parameters in the file and the database but this recommendation is not enforced by the Kea servers. In particular, if the subnets' configuration is sourced from the database, we recommend that all subnets are specified in the database and no subnets are specified in the configuration file. It is possible to specify the subnets in both places, but that must be done with care. The subnets in the configuration file with overlapping ids and/or prefixes with the subnets from the database will be superseded by those from the database.

Once the Kea server is configured, it starts periodically polling for the configuration changes in the database. The frequency of polling is controlled by the **config-fetch-wait-time** parameter. It is expressed in seconds and it is the period between the time when the server completed last polling (and possibly the local configuration update) and the time when it begins polling again. In the example above, this period is set to 20 seconds. This means that after adding a new configuration into the database (e.g. added new subnet), it will take up to 20 seconds (plus the time needed to fetch and apply the new configuration) before the server starts using this subnet. The lower the **config-fetch-wait-time** value, the shorter the time for the server to react to the incremental configuration updates in the database. On the other hand, polling the database too frequently may impact the DHCP server's performance because the server needs to make at least one query to the database to discover the pending configuration updates. The default value of the **config-fetch-wait-time** is 30 seconds.

Finally, in the configuration example above, two hooks libraries are loaded. The former, `libdhcp_mysql_cb.so`, is the implementation of the Configuration Backend for MySQL. It must be always present when the server uses MySQL as the configuration repository. Failing to load this library will result in an error during the server configuration if the "mysql" database is selected with the **config-control** parameter.



The latter hooks library, `libdhcp_cb_cmds.so`, is optional. It should be loaded when the Kea server instance is to be used for managing the configuration in the database. See the Section [15.4.8](#) for the details. Note that this hooks library is only available to the ISC customers with a support contract.



Chapter 9

The DHCPv6 Server

9.1 Starting and Stopping the DHCPv6 Server

It is recommended that the Kea DHCPv6 server be started and stopped using **keactrl** (described in Chapter 6); however, it is also possible to run the server directly. It accepts the following command-line switches:

- **-c file** - specifies the configuration file. This is the only mandatory switch.
- **-d** - specifies whether the server logging should be switched to verbose mode. In verbose mode, the logging severity and debuglevel specified in the configuration file are ignored and "debug" severity and the maximum debuglevel (99) are assumed. The flag is convenient for temporarily switching the server into maximum verbosity, e.g. when debugging.
- **-p server-port** - specifies the local UDP port on which the server will listen. This is only useful during testing, as a DHCPv6 server listening on ports other than the standard ones will not be able to handle regular DHCPv6 queries.
- **-P client-port** - specifies the remote UDP port to which the server will send all responses. This is only useful during testing, as a DHCPv6 server sending responses to ports other than the standard ones will not be able to handle regular DHCPv6 queries.
- **-t file** - specifies a configuration file to be tested. Kea-dhcp6 will load it, check it, and exit. During the test, log messages are printed to standard output and error messages to standard error. The result of the test is reported through the exit code (0 = configuration looks ok, 1 = error encountered). The check is not comprehensive; certain checks are possible only when running the server.
- **-v** - displays the Kea version and exits.
- **-V** - displays the Kea extended version with additional parameters and exits. The listing includes the versions of the libraries dynamically linked to Kea.
- **-W** - displays the Kea configuration report and exits. The report is a copy of the `config.report` file produced by `./configure`; it is embedded in the executable binary.

On startup, the server will detect available network interfaces and will attempt to open UDP sockets on all interfaces mentioned in the configuration file. Since the DHCPv6 server opens privileged ports, it requires root access. Make sure you run this daemon as root.

During startup, the server will attempt to create a PID file of the form: `[runstatedir]/kea/[conf name].kea-dhcp6.pid` where:

- **runstatedir**: The value as passed into the build configure script; it defaults to `"/usr/local/var/run"`. Note that this value may be overridden at runtime by setting the environment variable `KEA_PIDFILE_DIR`, although this is intended primarily for testing purposes.
- **conf name**: The configuration file name used to start the server, minus all preceding paths and the file extension. For example, given a pathname of `"/usr/local/etc/kea/myconf.txt"`, the portion used would be `"myconf"`.



If the file already exists and contains the PID of a live process, the server will issue a `DHCP6_ALREADY_RUNNING` log message and exit. It is possible, though unlikely, that the file is a remnant of a system crash and the process to which the PID belongs is unrelated to Kea. In such a case it would be necessary to manually delete the PID file.

The server can be stopped using the `kill` command. When running in a console, the server can also be shut down by pressing `ctrl-c`. It detects the key combination and shuts down gracefully.

9.2 DHCPv6 Server Configuration

9.2.1 Introduction

This section explains how to configure the DHCPv6 server using a configuration file. Before DHCPv6 is started, its configuration file has to be created. The basic configuration is as follows:

```
{
# DHCPv6 configuration starts on the next line
"Dhcp6": {

# First we set up global values
  "valid-lifetime": 4000,
  "renew-timer": 1000,
  "rebind-timer": 2000,
  "preferred-lifetime": 3000,

# Next we setup the interfaces to be used by the server.
  "interfaces-config": {
    "interfaces": [ "eth0" ]
  },

# And we specify the type of lease database
  "lease-database": {
    "type": "memfile",
    "persist": true,
    "name": "/var/lib/kea/dhcp6.leases"
  },

# Finally, we list the subnets from which we will be leasing addresses.
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "pools": [
        {
          "pool": "2001:db8:1::1-2001:db8:1::ffff"
        }
      ]
    }
  ]
}
# DHCPv6 configuration ends with the next line
}
```

The following paragraphs provide a brief overview of the parameters in the above example, along with their format. Subsequent sections of this chapter go into much greater detail for these and other parameters.

The lines starting with a hash (`#`) are comments and are ignored by the server; they do not impact its operation in any way.

The configuration starts in the first line with the initial opening curly bracket (or brace). Each configuration must contain an object specifying the configuration of the Kea module using it. In the example above this object is called **Dhcp6**.

**Note**

In the current Kea release it is possible to specify configurations of multiple modules within a single configuration file, but this is not recommended and support for it will be removed in the future releases. The only object, besides the one specifying module configuration, which can (and usually was) included in the same file is **Logging**. However, we don't include this object in the example above for clarity and its content, the list of loggers, should now be inside the **Dhcp4** object instead of this deprecated object.

The Dhcp6 configuration starts with the **"Dhcp6": {** line and ends with the corresponding closing brace (in the above example, the brace after the last comment). Everything defined between those lines is considered to be the Dhcp6 configuration.

In the general case, the order in which those parameters appear does not matter, but there are two caveats. The first one is to remember that the configuration file must be well-formed JSON. That means that the parameters for any given scope must be separated by a comma and there must not be a comma after the last parameter. When reordering a configuration file, keep in mind that moving a parameter to or from the last position in a given scope may also require moving the comma. The second caveat is that it is uncommon — although legal JSON — to repeat the same parameter multiple times. If that happens, the last occurrence of a given parameter in a given scope is used, while all previous instances are ignored. This is unlikely to cause any confusion as there are no real-life reasons to keep multiple copies of the same parameter in your configuration file.

Moving onto the DHCPv6 configuration elements, the first few elements define some global parameters. **valid-lifetime** defines how long the addresses (leases) given out by the server are valid. If nothing changes, a client that got an address is allowed to use it for 4000 seconds. (Note that integer numbers are specified as is, without any quotes around them.) The address will become deprecated in 3000 seconds, i.e. clients are allowed to keep old connections, but can't use this address for creating new connections. **renew-timer** and **rebind-timer** are values that define T1 and T2 timers that govern when the client will begin the renewal and rebind procedures.

The **interfaces-config** map specifies the server configuration concerning the network interfaces, on which the server should listen to the DHCP messages. The **interfaces** parameter specifies a list of network interfaces on which the server should listen. Lists are opened and closed with square brackets, with elements separated by commas. To listen on two interfaces, the **interfaces-config** should look like this:

```
"interfaces-config": {
  "interfaces": [ "eth0", "eth1" ]
},
```

The next couple of lines define the lease database, the place where the server stores its lease information. This particular example tells the server to use **memfile**, which is the simplest (and fastest) database backend. It uses an in-memory database and stores leases on disk in a CSV file. This is a very simple configuration; usually the lease database configuration is more extensive and contains additional parameters. Note that **lease-database** is an object and opens up a new scope, using an opening brace. Its parameters (just one in this example - **type**) follow. If there were more than one, they would be separated by commas. This scope is closed with a closing brace. As more parameters for the Dhcp6 definition follow, a trailing comma is present.

Finally, we need to define a list of IPv6 subnets. This is the most important DHCPv6 configuration structure as the server uses that information to process clients' requests. It defines all subnets from which the server is expected to receive DHCP requests. The subnets are specified with the **subnet6** parameter. It is a list, so it starts and ends with square brackets. Each subnet definition in the list has several attributes associated with it, so it is a structure and is opened and closed with braces. At a minimum, a subnet definition has to have at least two parameters: **subnet** (which defines the whole subnet) and **pools** (which is a list of dynamically allocated pools that are governed by the DHCP server).

The example contains a single subnet. If more than one were defined, additional elements in the **subnet6** parameter would be specified and separated by commas. For example, to define two subnets, the following syntax would be used:

```
"subnet6": [
  {
    "pools": [ { "pool": "2001:db8:1::/112" } ],
    "subnet": "2001:db8:1::/64"
  },
  {
    "pools": [ { "pool": "2001:db8:2::1-2001:db8:2::ffff" } ],
    "subnet": "2001:db8:2::/64"
  }
]
```



Note that indentation is optional and is used for aesthetic purposes only. In some cases it may be preferable to use more compact notation.

After all parameters are specified, we have two contexts open: global and Dhcp6, hence we need two closing curly brackets to close them.

9.2.2 Lease Storage

All leases issued by the server are stored in the lease database. Currently there are four database backends available: memfile (which is the default backend), MySQL, PostgreSQL, and Cassandra.

9.2.2.1 Memfile - Basic Storage for Leases

The server is able to store lease data in different repositories. Larger deployments may elect to store leases in a database. Section 9.2.2.2 describes this option. In typical smaller deployments though, the server will store lease information in a CSV file rather than a database. As well as requiring less administration, an advantage of using a file for storage is that it eliminates a dependency on third-party database software.

The configuration of the file backend (Memfile) is controlled through the Dhcp6/lease-database parameters. The **type** parameter is mandatory and it specifies which storage for leases the server should use. The value of "**memfile**" indicates that the file should be used as the storage. The following list gives additional optional parameters that can be used to configure the Memfile backend.

- **persist**: controls whether the new leases and updates to existing leases are written to the file. It is strongly recommended that the value of this parameter is set to **true** at all times, during the server's normal operation. Not writing leases to disk means that if a server is restarted (e.g. after a power failure), it will not know what addresses have been assigned. As a result, it may hand out addresses to new clients that are already in use. The value of **false** is mostly useful for performance-testing purposes. The default value of the **persist** parameter is **true**, which enables writing lease updates to the lease file.
- **name**: specifies an absolute location of the lease file in which new leases and lease updates will be recorded. The default value for this parameter is "`[kea-install-dir]/var/lib/kea/kea-leases6.csv`".
- **lfc-interval**: specifies the interval, in seconds, at which the server will perform a lease file cleanup (LFC). This removes redundant (historical) information from the lease file and effectively reduces the lease file size. The cleanup process is described in more detailed fashion further in this section. The default value of the **lfc-interval** is **3600**. A value of 0 disables the LFC.

An example configuration of the Memfile backend is presented below:

```
"Dhcp6": {
  "lease-database": {
    "type": "memfile",
    "persist": true,
    "name": "/tmp/kea-leases6.csv",
    "lfc-interval": 1800
  }
}
```

This configuration selects the `/tmp/kea-leases6.csv` as the storage for lease information and enables persistence (writing lease updates to this file). It also configures the backend to perform a periodic cleanup of the lease file every 30 minutes.

It is important to know how the lease file contents are organized to understand why the periodic lease file cleanup is needed. Every time the server updates a lease or creates a new lease for the client, the new lease information must be recorded in the lease file. For performance reasons, the server does not update the existing client's lease in the file, as this would potentially require rewriting the entire file. Instead, it simply appends the new lease information to the end of the file; the previous lease entries for the client are not removed. When the server loads leases from the lease file, e.g. at the server startup, it assumes that the latest lease entry for the client is the valid one. The previous entries are discarded, meaning that the server can re-construct the accurate information about the leases even though there may be many lease entries for each client. However, storing many entries for each client results in a bloated lease file and impairs the performance of the server's startup and reconfiguration, as it needs to process a larger number of lease entries.



Lease file cleanup (LFC) removes all previous entries for each client and leaves only the latest ones. The interval at which the cleanup is performed is configurable, and it should be selected according to the frequency of lease renewals initiated by the clients. The more frequent the renewals, the smaller the value of **lfc-interval** should be. Note, however, that the LFC takes time and thus it is possible (although unlikely) that, if the **lfc-interval** is too short, a new cleanup may be started while the previous one is still running. The server would recover from this by skipping the new cleanup when it detects that the previous cleanup is still in progress. But it implies that the actual cleanups will be triggered more rarely than configured. Moreover, triggering a new cleanup adds overhead to the server, which will not be able to respond to new requests for a short period of time when the new cleanup process is spawned. Therefore, it is recommended that the **lfc-interval** value is selected in a way that would allow for the LFC to complete the cleanup before a new cleanup is triggered.

Lease file cleanup is performed by a separate process (in the background) to avoid a performance impact on the server process. To avoid the conflicts between two processes both using the same lease files, the LFC process starts with Kea opening new lease file and the actual LFC process operates on the lease file that is no longer used by the server. There are also other files created as a side effect of the lease file cleanup. The detailed description of the LFC is located later in this Kea Administrator's Reference Manual: Chapter 13.

9.2.2.2 Lease Database Configuration

Note

Lease database access information must be configured for the DHCPv6 server, even if it has already been configured for the DHCPv4 server. The servers store their information independently, so each server can use a separate database or both servers can use the same database.

Lease database configuration is controlled through the `Dhcp6/lease-database` parameters. The type of the database must be set to "memfile", "mysql", "postgresql", or "cql", e.g.:

```
"Dhcp6": { "lease-database": { "type": "mysql", ... }, ... }
```

Next, the name of the database to hold the leases must be set; this is the name used when the database was created (see Section 4.3.2.1, Section 4.3.3.1, or Section 4.3.4.1).

```
"Dhcp6": { "lease-database": { "name": "database-name" , ... }, ... }
```

For Cassandra:

```
"Dhcp6": { "lease-database": { "keyspace": "database-name" , ... }, ... }
```

If the database is located on a different system from the DHCPv6 server, the database host name must also be specified. (It should be noted that this configuration may have a severe impact on server performance.):

```
"Dhcp6": { "lease-database": { "host": "remote-host-name", ... }, ... }
```

For Cassandra, multiple contact points can be provided:

```
"Dhcp6": { "lease-database": { "contact-points": "remote-host-name[, ...]" , ... }, ... }
```

Normally, the database will be on the same machine as the DHCPv6 server. In this case, set the value to the empty string:

```
"Dhcp6": { "lease-database": { "host" : "", ... }, ... }
```

For Cassandra:

```
"Dhcp6": { "lease-database": { "contact-points": "", ... }, ... }
```

Should the database use a port different than the default, it may be specified as well:

```
"Dhcp6": { "lease-database": { "port" : 12345, ... }, ... }
```



Should the database be located on a different system, you may need to specify a longer interval for the connection timeout:

```
"Dhcp6": { "lease-database": { "connect-timeout" : timeout-in-seconds, ... }, ... }
```

The default value of five seconds should be more than adequate for local connections. If a timeout is given, though, it should be an integer greater than zero.

The maximum number of times the server will automatically attempt to reconnect to the lease database after connectivity has been lost may be specified:

```
"Dhcp6": { "lease-database": { "max-reconnect-tries" : number-of-tries, ... }, ... }
```

If the server is unable to reconnect to the database after making the maximum number of attempts the server will exit. A value of zero (the default) disables automatic recovery and the server will exit immediately upon detecting a loss of connectivity (MySQL and Postgres only).

The number of milliseconds the server will wait in between attempts to reconnect to the lease database after connectivity has been lost may also be specified:

```
"Dhcp6": { "lease-database": { "reconnect-wait-time" : number-of-milliseconds, ... }, ... }
```

The default value for MySQL and Postgres is 0, which disables automatic recovery and causes the server to exit immediately upon detecting the loss of connectivity. The default value for Cassandra is 2000 ms.

Note

Automatic reconnection to database backends is configured individually per backend. This allows you to tailor the recovery parameters to each backend you use. We do suggest that you enable it either for all backends or no backends so you have consistent behavior. Losing connectivity to a backend for which reconnect is disabled will result in the server shutting itself down. This includes cases when the lease database backend and the hosts database backend are connected to the same database instance.

Note Note that host parameter is used by MySQL and PostgreSQL backends. Cassandra has a concept of contact points that could be used to contact the cluster, instead of a single IP or hostname. It takes a list of comma-separated IP addresses, which may be specified as:

```
"Dhcp6": { "lease-database": { "contact-points" : "192.0.2.1,192.0.2.2", ... }, ... }
```

Finally, the credentials of the account under which the server will access the database should be set:

```
"Dhcp6": { "lease-database": { "user": "user-name",  
                             "password": "password",  
                             ... },  
  ... }
```

If there is no password to the account, set the password to the empty string "". (This is also the default.)

9.2.2.3 Cassandra-Specific Parameters

The parameters are the same for DHCPv4 and DHCPv6. See Section [8.2.2.3](#) for details.



9.2.3 Hosts Storage

Kea is also able to store information about host reservations in the database. The hosts database configuration uses the same syntax as the lease database. In fact, a Kea server opens independent connections for each purpose, be it lease or hosts information. This arrangement gives the most flexibility. Kea can keep leases and host reservations separately, but can also point to the same database. Currently the supported hosts database types are MySQL, PostgreSQL and Cassandra.

For example, the following configuration can be used to configure connection to MySQL:

```
"Dhcp6": {
  "hosts-database": {
    "type": "mysql",
    "name": "kea",
    "user": "kea",
    "password": "secret123",
    "host": "localhost",
    "port": 3306
  }
}
```

Note that depending on your database configuration, many of the parameters may be optional.

Please note that usage of hosts storage is optional. A user can define all host reservations in the configuration file, and that is the recommended way if the number of reservations is small. However, when the number of reservations grows, it is more convenient to use host storage. Please note that both storage methods (configuration file and one of the supported databases) can be used together. If hosts are defined in both places, the definitions from the configuration file are checked first and external storage is checked later, if necessary.

In fact, host information can be placed in multiple stores. Operations are performed on the stores in the order they are defined in the configuration file, although this leads to a restriction in ordering in the case of a host reservation addition; read-only stores must be configured after a (required) read-write store, or the addition will fail.

9.2.3.1 DHCPv6 Hosts Database Configuration

Hosts database configuration is controlled through the Dhcp6/hosts-database parameters. If enabled, the type of database must be set to "mysql" or "postgresql".

```
"Dhcp6": { "hosts-database": { "type": "mysql", ... }, ... }
```

Next, the name of the database to hold the reservations must be set; this is the name used when the database was created (see Section 4.3 for instructions on how to set up the desired database type).

```
"Dhcp6": { "hosts-database": { "name": "database-name" , ... }, ... }
```

If the database is located on a different system than the DHCPv6 server, the database host name must also be specified. (Again it should be noted that this configuration may have a severe impact on server performance.)

```
"Dhcp6": { "hosts-database": { "host": remote-host-name, ... }, ... }
```

Normally, the database will be on the same machine as the DHCPv6 server. In this case, set the value to the empty string:

```
"Dhcp6": { "hosts-database": { "host" : "", ... }, ... }
```

```
"Dhcp6": { "hosts-database": { "port" : 12345, ... }, ... }
```

The maximum number of times the server will automatically attempt to reconnect to the host database after connectivity has been lost may be specified:

```
"Dhcp6": { "host-database": { "max-reconnect-tries" : number-of-tries, ... }, ... }
```



If the server is unable to reconnect to the database after making the maximum number of attempts the server will exit. A value of zero (the default) disables automatic recovery and the server will exit immediately upon detecting a loss of connectivity (MySQL and Postgres only). For Cassandra, Kea uses a Cassandra interface that connects to all nodes in a cluster at the same time. Any connectivity issues should be handled by internal Cassandra mechanisms.

The number of milliseconds the server will wait between attempts to reconnect to the host database after connectivity has been lost may also be specified:

```
"Dhcp6": { "hosts-database": { "reconnect-wait-time" : number-of-milliseconds, ... }, ... }
```

The default value for MySQL and Postgres is 0, which disables automatic recovery and causes the server to exit immediately upon detecting the loss of connectivity. The default value for Cassandra is 2000 ms.

Note

Automatic reconnection to database backends is configured individually per backend. This allows you to tailor the recovery parameters to each backend you use. We do suggest that you enable it either for all backends or no backends so you have consistent behavior. Losing connectivity to a backend for which reconnect is disabled will result in the server shutting itself down. This includes cases when the lease database backend and the hosts database backend are connected to the same database instance.

Finally, the credentials of the account under which the server will access the database should be set:

```
"Dhcp6": { "hosts-database": { "user": "user-name",  
                             "password": "password",  
                             ... },  
          ... }
```

If there is no password to the account, set the password to the empty string "". (This is also the default.)

The multiple store extension uses a similar syntax; a configuration is placed into a "hosts-databases" list instead of into a "hosts-database" entry as in:

```
"Dhcp6": { "hosts-databases": [ { "type": "mysql", ... }, ... ], ... }
```

For additional Cassandra-specific parameters, see Section [8.2.2.3](#).

9.2.3.2 Using Read-Only Databases for Host Reservations

In some deployments the database user whose name is specified in the database backend configuration may not have write privileges to the database. This is often required by the policy within a given network to secure the data from being unintentionally modified. In many cases administrators have deployed inventory databases, which contain substantially more information about the hosts than just the static reservations assigned to them. The inventory database can be used to create a view of a Kea hosts database and such a view is often read-only.

Kea host database backends operate with an implicit configuration to both read from and write to the database. If the database user does not have write access to the host database, the backend will fail to start and the server will refuse to start (or reconfigure). However, if access to a read-only host database is required for retrieving reservations for clients and/or assigning specific addresses and options, it is possible to explicitly configure Kea to start in "read-only" mode. This is controlled by the **readonly** boolean parameter as follows:

```
"Dhcp6": { "hosts-database": { "readonly": true, ... }, ... }
```

Setting this parameter to **false** configures the database backend to operate in "read-write" mode, which is also the default configuration if the parameter is not specified.

Note

The **readonly** parameter is currently only supported for MySQL and PostgreSQL databases.



9.2.4 Interface Configuration

The DHCPv6 server has to be configured to listen on specific network interfaces. The simplest network interface configuration instructs the server to listen on all available interfaces:

```
"Dhcp6": {
  "interfaces-config": {
    "interfaces": [ "*" ]
  }
  ...
}
```

The asterisk plays the role of a wildcard and means "listen on all interfaces." However, it is usually a good idea to explicitly specify interface names:

```
"Dhcp6": {
  "interfaces-config": {
    "interfaces": [ "eth1", "eth3" ]
  },
  ...
}
```

It is possible to use a wildcard interface name (asterisk) concurrently with explicit interface names:

```
"Dhcp6": {
  "interfaces-config": {
    "interfaces": [ "eth1", "eth3", "*" ]
  },
  ...
}
```

It is anticipated that this form of usage will only be used when it is desired to temporarily override a list of interface names and listen on all interfaces.

As with the DHCPv4 server, binding to specific addresses and disabling re-detection of interfaces are supported. But **dhcp-socket-type** is not because DHCPv6 uses UDP/IPv6 sockets only. The following example shows how to disable the interface detection:

```
"Dhcp6": {
  "interfaces-config": {
    "interfaces": [ "eth1", "eth3" ],
    "re-detect": false
  },
  ...
}
```

The loopback interfaces (i.e. the "lo" or "lo0" interface) are not configured by default, unless explicitly mentioned in the configuration. Note that Kea requires a link-local address (which does not exist on all systems), or a specified unicast address as in:

```
"Dhcp6": {
  "interfaces-config": {
    "interfaces": [ "enp0s2/2001:db8::1234:abcd" ]
  },
  ...
}
```

9.2.5 IPv6 Subnet Identifier

The subnet identifier is a unique number associated with a particular subnet. In principle, it is used to associate clients' leases with their respective subnets. When a subnet identifier is not specified for a subnet being configured, it will be automatically assigned



by the configuration mechanism. The identifiers are assigned from 1 and are monotonically increased for each subsequent subnet: 1, 2, 3

If there are multiple subnets configured with auto-generated identifiers and one of them is removed, the subnet identifiers may be renumbered. For example: if there are four subnets and the third is removed, the last subnet will be assigned the identifier that the third subnet had before removal. As a result, the leases stored in the lease database for subnet 3 are now associated with subnet 4, something that may have unexpected consequences. The only remedy for this issue at present is to manually specify a unique identifier for each subnet.

Note Subnet IDs must be greater than zero and less than 4294967295.

The following configuration will assign the specified subnet identifier to the newly configured subnet:

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "id": 1024,
      ...
    }
  ]
}
```

This identifier will not change for this subnet unless the "id" parameter is removed or set to 0. The value of 0 forces auto-generation of the subnet identifier.

9.2.6 IPv6 Subnet Prefix

The subnet prefix is the second way to identify a subnet. It does not need to have the address part to match the prefix length, for instance this configuration is accepted:

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:1::1/64",
      ...
    }
  ]
}
```

Even there is another subnet with the "2001:db8:1::/64" prefix: only the textual form of subnets are compared to avoid duplicates.

Note Abuse of this feature can lead to incorrect subnet selection (see Section 9.2.18).

9.2.7 Unicast Traffic Support

When the DHCPv6 server starts, by default it listens to the DHCP traffic sent to multicast address ff02::1:2 on each interface that it is configured to listen on (see Section 9.2.4). In some cases it is useful to configure a server to handle incoming traffic sent to global unicast addresses as well. The most common reason for this is to have relays send their traffic to the server directly. To configure the server to listen on a specific unicast address, the interface name can be optionally followed by a slash, followed by the global unicast address on which the server should listen. The server listens to this address in addition to normal link-local binding and listening on the ff02::1:2 address. The sample configuration below shows how to listen on 2001:db8::1 (a global address) configured on the eth1 interface.



```
"Dhcp6": {
  "interfaces-config": {
    "interfaces": [ "eth1/2001:db8::1" ]
  },
  ...
  "option-data": [
    {
      "name": "unicast",
      "data": "2001:db8::1"
    } ],
  ...
}
```

This configuration will cause the server to listen on eth1 on the link-local address, the multicast group (ff02::1:2), and 2001:db8::1.

Usually unicast support is associated with a server unicast option which allows clients to send unicast messages to the server. The example above includes a server unicast option specification which will cause the client to send messages to the specified unicast address.

It is possible to mix interface names, wildcards, and interface names/addresses in the list of interfaces. It is not possible, however, to specify more than one unicast address on a given interface.

Care should be taken to specify proper unicast addresses. The server will attempt to bind to the addresses specified without any additional checks. This approach was selected on purpose to allow the software to communicate over uncommon addresses if so desired.

9.2.8 Subnet and Address Pool

The main role of a DHCPv6 server is address assignment. For this, the server must be configured with at least one subnet and one pool of dynamic addresses to be managed. For example, assume that the server is connected to a network segment that uses the 2001:db8:1::/64 prefix. The administrator of that network decides that addresses from range 2001:db8:1::1 to 2001:db8:1::ffff are going to be managed by the Dhcp6 server. Such a configuration can be achieved in the following way:

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "pools": [
        {
          "pool": "2001:db8:1::1-2001:db8:1::ffff"
        }
      ]
    },
    ...
  ]
}
```

Note that **subnet** is defined as a simple string, but the **pools** parameter is actually a list of pools; for this reason, the pool definition is enclosed in square brackets, even though only one range of addresses is specified.

Each **pool** is a structure that contains the parameters that describe a single pool. Currently there is only one parameter, **pool**, which gives the range of addresses in the pool.

It is possible to define more than one pool in a subnet; continuing the previous example, further assume that 2001:db8:1:0:5::/80 should also be managed by the server. It could be written as 2001:db8:1:0:5:: to 2001:db8:1:0:5::ffff:ffff:ffff, but typing so many 'f's is cumbersome. It can be expressed more simply as 2001:db8:1:0:5::/80. Both formats are supported by Dhcp6 and can be mixed in the pool list. For example, one could define the following pools:

```
"Dhcp6": {
  "subnet6": [
    {
```



```
"subnet": "2001:db8:1::/64",
"pools": [
  { "pool": "2001:db8:1::1-2001:db8:1::ffff" },
  { "pool": "2001:db8:1:05::/80" }
],
...
}
]
```

White space in pool definitions is ignored, so spaces before and after the hyphen are optional. They can be used to improve readability.

The number of pools is not limited, but for performance reasons it is recommended to use as few as possible.

The server may be configured to serve more than one subnet. To add a second subnet, use a command similar to the following:

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "pools": [
        { "pool": "2001:db8:1::1-2001:db8:1::ffff" }
      ]
    },
    {
      "subnet": "2001:db8:2::/64",
      "pools": [
        { "pool": "2001:db8:2::/64" }
      ]
    },
    ...
  ]
}
```

In this example, we allow the server to dynamically assign all addresses available in the whole subnet. Although rather wasteful, it is certainly a valid configuration to dedicate the whole /64 subnet for that purpose. Note that the Kea server does not preallocate the leases, so there is no danger in using gigantic address pools.

When configuring a DHCPv6 server using prefix/length notation, please pay attention to the boundary values. When specifying that the server can use a given pool, it will also be able to allocate the first (typically a network address) address from that pool. For example, for pool 2001:db8:2::/64 the 2001:db8:2:: address may be assigned as well. To avoid this, use the "min-max" notation.

9.2.9 Subnet and Prefix Delegation Pools

Subnets may also be configured to delegate prefixes, as defined in [RFC 8415](#), section 6.3. A subnet may have one or more prefix delegation pools. Each pool has a prefixed address, which is specified as a prefix (**prefix**) and a prefix length (**prefix-len**), as well as a delegated prefix length (**delegated-len**). The delegated length must not be shorter than (that is, it must be numerically greater than or equal to) the prefix length. If both the delegated and prefix lengths are equal, the server will be able to delegate only one prefix. The delegated prefix does not have to match the subnet prefix.

Below is a sample subnet configuration which enables prefix delegation for the subnet:

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:d8b:1::/64",
      "pd-pools": [
        {
```



```
        "prefix": "3000:1::",
        "prefix-len": 64,
        "delegated-len": 96
    }
]
},
...
}
```

9.2.10 Prefix Exclude Option

For each delegated prefix, the delegating router may choose to exclude a single prefix out of the delegated prefix as specified in [RFC 6603](#). The requesting router must not assign the excluded prefix to any of its downstream interfaces, and it is intended to be used on a link through which the delegating router exchanges DHCPv6 messages with the requesting router. The configuration example below demonstrates how to specify an excluded prefix within a prefix pool definition. The excluded prefix "2001:db8:1:8000:cafe:80::/72" will be sent to a requesting router which includes the Prefix Exclude option in the ORO, and which is delegated a prefix from this pool.

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:1::/48",
      "pd-pools": [
        {
          "prefix": "2001:db8:1:8000::",
          "prefix-len": 48,
          "delegated-len": 64,
          "excluded-prefix": "2001:db8:1:8000:cafe:80::",
          "excluded-prefix-len": 72
        }
      ]
    }
  ]
}
```

9.2.11 Standard DHCPv6 Options

One of the major features of a DHCPv6 server is the ability to provide configuration options to clients. Although there are several options that require special behavior, most options are sent by the server only if the client explicitly requests them. The following example shows how to configure one of the most frequently used options, which supplies the address of DNS servers. Options specified in this way are considered global and apply to all configured subnets.

```
"Dhcp6": {
  "option-data": [
    {
      "name": "dns-servers",
      "code": 23,
      "space": "dhcp6",
      "csv-format": true,
      "data": "2001:db8::cafe, 2001:db8::babe"
    },
    ...
  ]
}
```

The **option-data** line creates a new entry in the option-data table. This table contains information on all global options that the server is supposed to configure in all subnets. The **name** line specifies the option name. (For a complete list of currently



supported names, see Table 9.1.) The next line specifies the option code, which must match one of the values from that list. The line beginning with **space** specifies the option space, which must always be set to "dhcp6" as these are standard DHCPv6 options. For other name spaces, including custom option spaces, see Section 9.2.15. The following line specifies the format in which the data will be entered; use of CSV (comma-separated values) is recommended. Finally, the **data** line gives the actual value to be sent to clients. Data is specified as normal text, with values separated by commas if more than one value is allowed.

Options can also be configured as hexadecimal values. If "csv-format" is set to false, the option data must be specified as a hexadecimal string. The following commands configure the DNS-SERVERS option for all subnets with the following addresses: 2001:db8:1::cafe and 2001:db8:1::babe.

```
"Dhcp6": {
  "option-data": [
    {
      "name": "dns-servers",
      "code": 23,
      "space": "dhcp6",
      "csv-format": false,
      "data": "20 01 0D B8 00 01 00 00 00 00 00 00 00 00 CA FE
              20 01 0D B8 00 01 00 00 00 00 00 00 00 00 BA BE"
    },
    ...
  ]
}
```

Note

The value for the setting of the "data" element is split across two lines in this example for clarity; when entering the command, the whole string should be entered on the same line.

Kea supports the following formats when specifying hexadecimal data:

- **Delimited octets** One or more octets separated by either colons or spaces (':' or ' '). While each octet may contain one or two digits, we strongly recommend always using two digits. Valid examples are "ab:cd:ef" and "ab cd ef".
- **String of digits** A continuous string of hexadecimal digits with or without a "0x" prefix. Valid examples are "0xabcdef" and "abcdef".

Care should be taken to use proper encoding when using hexadecimal format. Kea's ability to validate data correctness in hexadecimal is limited.

Most of the parameters in the "option-data" structure are optional and can be omitted in some circumstances as discussed in Section 9.2.16. Only one of name or code is required; you don't need to specify both. Space has a default value of "dhcp6", so you can skip this as well if you define a regular (not encapsulated) DHCPv6 option. Finally, csv-format defaults to true, so it too can be skipped, unless you want to specify the option value as hexstring. Therefore the above example can be simplified to:

```
"Dhcp6": {
  "option-data": [
    {
      "name": "dns-servers",
      "data": "2001:db8::cafe, 2001:db8::babe"
    },
    ...
  ]
}
```

Defined options are added to the response when the client requests them, as well as any options required by a protocol. An administrator can also specify that an option is always sent, even if a client did not specifically request it. To enforce the addition of a particular option, set the "always-send" flag to true as in:



```
"Dhcp6": {
  "option-data": [
    {
      "name": "dns-servers",
      "data": "2001:db8::cafe, 2001:db8::babe",
      "always-send": true
    },
    ...
  ]
}
```

The effect is the same as if the client added the option code in the Option Request Option (or its equivalent for vendor options) so in:

```
"Dhcp6": {
  "option-data": [
    {
      "name": "dns-servers",
      "data": "2001:db8::cafe, 2001:db8::babe",
      "always-send": true
    },
    ...
  ],
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "option-data": [
        {
          "name": "dns-servers",
          "data": "2001:db8:1::cafe, 2001:db8:1::babe"
        },
        ...
      ],
      ...
    },
    ...
  ],
  ...
}
```

The DNS servers option is always added to responses (the always-send is "sticky") but the value is the subnet one when the client is localized in the subnet.

It is possible to override options on a per-subnet basis. If clients connected to most of your subnets are expected to get the same values of a given option, you should use global options; you can then override specific values for a small number of subnets. On the other hand, if you use different values in each subnet, it does not make sense to specify global option values; rather, you should set only subnet-specific ones.

The following commands override the global DNS servers option for a particular subnet, setting a single DNS server with address 2001:db8:1::3.

```
"Dhcp6": {
  "subnet6": [
    {
      "option-data": [
        {
          "name": "dns-servers",
          "code": 23,
          "space": "dhcp6",
          "csv-format": true,
          "data": "2001:db8:1::3"
        }
      ]
    }
  ]
}
```



```
    },
    ...
  ],
  ...
},
...
]
```

In some cases it is useful to associate some options with an address or prefix pool from which a client is assigned a lease. Pool-specific option values override subnet-specific and global option values. If the client is assigned multiple leases from different pools, the server will assign options from all pools from which the leases have been obtained. However, if the particular option is specified in multiple pools from which the client obtains the leases, only one instance of this option will be handed out to the client. The server's administrator must not try to prioritize assignment of pool-specific options by trying to order pools declarations in the server configuration.

The following configuration snippet demonstrates how to specify the DNS servers option, which will be assigned to a client only if the client obtains an address from the given pool:

```
"Dhcp6": {
  "subnet6": [
    {
      "pools": [
        {
          "pool": "2001:db8:1::100-2001:db8:1::300",
          "option-data": [
            {
              "name": "dns-servers",
              "data": "2001:db8:1::10"
            }
          ]
        }
      ]
    },
    ...
  ],
  ...
}
```

Options can also be specified in a class of host reservation scope. The current Kea options precedence order is (from most important): host reservation, pool, subnet, shared network, class, global.

The currently supported standard DHCPv6 options are listed in Table 9.1. "Name" and "Code" are the values that should be used as a name/code in the option-data structures. "Type" designates the format of the data; the meanings of the various types is given in Table 8.2.

When a data field is a string, and that string contains the comma (,; U+002C) character, the comma must be escaped with double backslashes (\, U+005C). This double escape is required, because both the routine splitting CSV data into fields and JSON use the same escape character; a single escape (\,) would make the JSON invalid. For example, the string "EST5EDT4,M3.2.0/02:00,M11.1.0/02" would be represented as:

```
"Dhcp6": {
  "subnet6": [
    {
      "pools": [
        {
          "option-data": [
            {
              "name": "new-posix-timezone",
              "data": "EST5EDT4\\,M3.2.0/02:00\\,M11.1.0/02:00"
            }
          ]
        }
      ]
    }
  ]
}
```



```
        ]
        },
        ...
    ],
    ...
},
...
}
```

Some options are designated as arrays, which means that more than one value is allowed in such an option. For example, the option `dns-servers` allows the specification of more than one IPv6 address, enabling clients to obtain the addresses of multiple DNS servers.

Section 9.2.13 describes the configuration syntax to create custom option definitions (formats). Creation of custom definitions for standard options is generally not permitted, even if the definition being created matches the actual option format defined in the RFCs. There is an exception to this rule for standard options for which Kea currently does not provide a definition. In order to use such options, a server administrator must create a definition as described in Section 9.2.13 in the 'dhcp6' option space. This definition should match the option format described in the relevant RFC, but the configuration mechanism would allow any option format as it currently has no means to validate it.

Options marked with (1) have option definitions, but the logic behind them is not implemented. That means that technically Kea knows how to parse them in incoming messages or how to send them if configured to do so, but not what to do with them. Since the related RFCs require certain processing, the support for those options is non-functional. However, it may be useful in some limited lab testing; hence the definition formats are listed here.

9.2.12 Common Softwire46 Options

Softwire46 options are involved in IPv4 over IPv6 provisioning by means of tunneling or translation as specified in RFC 7598. The following sections provide configuration examples of these options.

9.2.12.1 Softwire46 Container Options

S46 container options group rules and optional port parameters for a specified domain. There are three container options specified in the "dhcp6" (top-level) option space: the MAP-E Container option, the MAP-T Container option, and the S46 Lightweight 4over6 Container option. These options only contain encapsulated options specified below; they do not include any data fields.

To configure the server to send a specific container option along with all encapsulated options, the container option must be included in the server configuration as shown below:

```
"Dhcp6": {
    ...
    "option-data": [
        {
            "name": "s46-cont-mape"
        } ],
    ...
}
```

This configuration will cause the server to include the MAP-E Container option to the client. Use "s46-cont-mapt" or "s46-cont-lw" for the MAP-T Container and S46 Lightweight 4over6 Container options, respectively.

All remaining Softwire options described below are included in one of the container options. Thus, they have to be included in appropriate option spaces by selecting a "space" name, which specifies in which option they are supposed to be included.



Name	Code	Type	Array?
preference	7	uint8	false
unicast	12	ipv6-address	false
vendor-opts	17	uint32	false
sip-server-dns	21	fqdn	true
sip-server-addr	22	ipv6-address	true
dns-servers	23	ipv6-address	true
domain-search	24	fqdn	true
nis-servers	27	ipv6-address	true
nisp-servers	28	ipv6-address	true
nis-domain-name	29	fqdn	true
nisp-domain-name	30	fqdn	true
sntp-servers	31	ipv6-address	true
information-refresh-time	32	uint32	false
bcmcs-server-dns	33	fqdn	true
bcmcs-server-addr	34	ipv6-address	true
geoconf-civic	36	record (uint8, uint16, binary)	false
remote-id	37	record (uint32, binary)	false
subscriber-id	38	binary	false
client-fqdn	39	record (uint8, fqdn)	false
pana-agent	40	ipv6-address	true
new-posix-timezone	41	string	false
new-tzdb-timezone	42	string	false
ero	43	uint16	true
lq-query (1)	44	record (uint8, ipv6-address)	false
client-data (1)	45	empty	false
clt-time (1)	46	uint32	false
lq-relay-data (1)	47	record (ipv6-address, binary)	false
lq-client-link (1)	48	ipv6-address	true
v6-lost	51	fqdn	false
capwap-ac-v6	52	ipv6-address	true
relay-id	53	binary	false
v6-access-domain	57	fqdn	false
sip-ua-cs-list	58	fqdn	true
bootfile-url	59	string	false
bootfile-param	60	tuple	true
client-arch-type	61	uint16	true
nii	62	record (uint8, uint8, uint8)	false
aftr-name	64	fqdn	false
erp-local-domain-name	65	fqdn	false
rsoo	66	empty	false
pd-exclude	67	binary	false
rdnss-selection	74	record (ipv6-address, uint8, fqdn)	true
client-linklayer-addr	79	binary	false
link-address	80	ipv6-address	false
solmax-rt	82	uint32	false
inf-max-rt	83	uint32	false
dhcp4o6-server-addr	88	ipv6-address	true
s46-rule	89	record (uint8, uint8, uint8, ipv4-address, ipv6-prefix)	false
s46-br	90	ipv6-address	false
s46-dmr	91	ipv6-prefix	false
s46-v4v6bind	92	record (ipv4-address, ipv6-prefix)	false
s46-portparams	93	record(uint8, psid)	false
s46-cont-mape	94	empty	false
s46-cont-mapt	95	empty	false
s46-cont-lw	96	empty	false
v6-captive-portal	103	string	false
ipv6-address-andsf	143	ipv6-address	true



9.2.12.2 S46 Rule Option

The S46 Rule option is used for conveying the Basic Mapping Rule (BMR) and Forwarding Mapping Rule (FMR).

```
{
  "space": "s46-cont-mape-options",
  "name": "s46-rule",
  "data": "128, 0, 24, 192.0.2.0, 2001:db8:1::/64"
}
```

Another possible "space" value is "s46-cont-mapt-options".

The S46 Rule option conveys a number of parameters:

- **flags**, an unsigned 8-bit integer, with currently only the most-significant bit specified. It denotes whether the rule can be used for forwarding (128) or not (0).
- **ea-len**, an 8-bit-long Embedded Address length. Allowed values range from 0 to 48.
- **IPv4 prefix length**, 8 bits long; expresses the prefix length of the Rule IPv4 prefix specified in the ipv4-prefix field. Allowed values range from 0 to 32.
- **IPv4 prefix**, a fixed-length 32-bit field that specifies the IPv4 prefix for the S46 rule. The bits in the prefix after prefix4-len number of bits are reserved, and MUST be initialized to zero by the sender and ignored by the receiver.
- **IPv6 prefix**, in prefix/length notation that specifies the IPv6 domain prefix for the S46 rule. The field is padded on the right with zero bits up to the nearest octet boundary, when prefix6-len is not evenly divisible by 8.

9.2.12.3 S46 BR Option

The S46 BR option is used to convey the IPv6 address of the Border Relay. This option is mandatory in the MAP-E Container option and is not permitted in the MAP-T and S46 Lightweight 4over6 Container options.

```
{
  "space": "s46-cont-mape-options",
  "name": "s46-br",
  "data": "2001:db8:cafe::1",
}
```

Another possible "space" value is "s46-cont-lw-options".

9.2.12.4 S46 DMR Option

The S46 DMR option is used to convey values for the Default Mapping Rule (DMR). This option is mandatory in the MAP-T container option and is not permitted in the MAP-E and S46 Lightweight 4over6 Container options.

```
{
  "space": "s46-cont-mapt-options",
  "name": "s46-dmr",
  "data": "2001:db8:cafe::/64",
}
```

This option must not be included in other containers.



9.2.12.5 S46 IPv4/IPv6 Address Binding option.

The S46 IPv4/IPv6 Address Binding option may be used to specify the full or shared IPv4 address of the Customer Edge (CE). The IPv6 prefix field is used by the CE to identify the correct prefix to use for the tunnel source.

```
{
  "space": "s46-cont-lw",
  "name": "s46-v4v6bind",
  "data": "192.0.2.3, 2001:db8:1:cafe::/64"
}
```

This option must not be included in other containers.

9.2.12.6 S46 Port Parameters

The S46 Port Parameters option specifies optional port-set information that MAY be provided to CEs.

```
{
  "space": "s46-rule-options",
  "name": "s46-portparams",
  "data": "2, 3/4",
}
```

Another possible "space" value is "s46-v4v6bind", to include this option in the S46 IPv4/IPv6 Address Binding option.

Note that the second value in the example above specifies the PSID and PSID-length fields in the format of PSID/PSID length. This is equivalent to the values of PSID-len=4 and PSID=12288 conveyed in the S46 Port Parameters option.

9.2.13 Custom DHCPv6 Options

It is possible to define options in addition to the standard ones. Assume that we want to define a new DHCPv6 option called "foo" which will have code 100 and which will convey a single, unsigned, 32-bit integer value. We can define such an option by using the following commands:

```
"Dhcp6": {
  "option-def": [
    {
      "name": "foo",
      "code": 100,
      "type": "uint32",
      "array": false,
      "record-types": "",
      "space": "dhcp6",
      "encapsulate": ""
    }, ...
  ],
  ...
}
```

The "false" value of the **array** parameter determines that the option does NOT comprise an array of "uint32" values but is, instead, a single value. Two other parameters have been left blank: **record-types** and **encapsulate**. The former specifies the comma-separated list of option data fields, if the option comprises a record of data fields. The **record-types** value should be non-empty if **type** is set to "record"; otherwise it must be left blank. The latter parameter specifies the name of the option space being encapsulated by the particular option. If the particular option does not encapsulate any option space, it should be left blank. Note that the above example only defines the format of the new option and does not set its value(s).

Only the **name**, **code**, and **type** parameters are required; all others are optional. The **array** default value is **false**. The **record-types** and **encapsulate** default values are blank (i.e. ""). The default **space** is "dhcp6".

Once the new option format is defined, its value is set in the same way as for a standard option. For example, the following commands set a global value that applies to all subnets.



```
"Dhcp6": {
  "option-data": [
    {
      "name": "foo",
      "code": 100,
      "space": "dhcp6",
      "csv-format": true,
      "data": "12345"
    }, ...
  ],
  ...
}
```

New options can take more complex forms than simple use of primitives (uint8, string, ipv6-address, etc); it is possible to define an option comprising a number of existing primitives.

For example, assume we want to define a new option that will consist of an IPv6 address, followed by an unsigned 16-bit integer, followed by a boolean value, followed by a text string. Such an option could be defined in the following way:

```
"Dhcp6": {
  "option-def": [
    {
      "name": "bar",
      "code": 101,
      "space": "dhcp6",
      "type": "record",
      "array": false,
      "record-types": "ipv6-address, uint16, boolean, string",
      "encapsulate": ""
    }, ...
  ],
  ...
}
```

The "type" is set to "record" to indicate that the option contains multiple values of different types. These types are given as a comma-separated list in the **record-types** field and should be ones from those listed in Table 8.2.

The values of the options are set in a **option-data** statement as follows:

```
"Dhcp6": {
  "option-data": [
    {
      "name": "bar",
      "space": "dhcp6",
      "code": 101,
      "csv-format": true,
      "data": "2001:db8:1::10, 123, false, Hello World"
    }
  ],
  ...
}
```

csv-format is set to **true** to indicate that the **data** field comprises a command-separated list of values. The values in **data** must correspond to the types set in the **record-types** field of the option definition.

When **array** is set to **true** and **type** is set to "record", the last field is an array, i.e. it can contain more than one value, as in:

```
"Dhcp6": {
  "option-def": [
    {
      "name": "bar",
      "code": 101,
```



```
        "space": "dhcp6",
        "type": "record",
        "array": true,
        "record-types": "ipv6-address, uint16",
        "encapsulate": ""
    }, ...
],
...
}
```

The new option content is one IPv6 address followed by one or more 16-bit unsigned integers.

Note

In general, boolean values are specified as **true** or **false**, without quotes. Some specific boolean parameters may accept also **"true"**, **"false"**, **0**, **1**, **"0"**, and **"1"**.

9.2.14 DHCPv6 Vendor-Specific Options

Currently there are two option spaces defined for the DHCPv6 daemon: "dhcp6" (for top-level DHCPv6 options) and "vendor-opts-space", which is empty by default, but in which options can be defined. Those options are carried in the Vendor-Specific Information option (code 17). The following examples show how to define an option "foo" with code 1 that consists of an IPv6 address, an unsigned 16-bit integer, and a string. The "foo" option is conveyed in a Vendor-Specific Information option, which comprises a single uint32 value that is set to "12345". The sub-option "foo" follows the data field holding this value.

```
"Dhcp6": {
  "option-def": [
    {
      "name": "foo",
      "code": 1,
      "space": "vendor-opts-space",
      "type": "record",
      "array": false,
      "record-types": "ipv6-address, uint16, string",
      "encapsulate": ""
    }
  ],
  ...
}
```

(Note that the option space is set to **vendor-opts-space**.) Once the option format is defined, the next step is to define actual values for that option:

```
"Dhcp6": {
  "option-data": [
    {
      "name": "foo",
      "space": "vendor-opts-space",
      "data": "2001:db8:1::10, 123, Hello World"
    }
  ],
  ...
},
...
}
```

We should also define a value (enterprise-number) for the Vendor-Specific Information option, that conveys our option "foo".

```
"Dhcp6": {
  "option-data": [
```



```
    ...  
    {  
      "name": "vendor-opts",  
      "data": "12345"  
    }  
  ],  
  ...  
}
```

Alternatively, the option can be specified using its code.

```
"Dhcp6": {  
  "option-data": [  
    ...  
    {  
      "code": 17,  
      "data": "12345"  
    }  
  ],  
  ...  
}
```

9.2.15 Nested DHCPv6 Options (Custom Option Spaces)

It is sometimes useful to define completely new option spaces. This is the case when a user wants their new option to convey sub-options that use a separate numbering scheme, for example sub-options with codes 1 and 2. Those option codes conflict with standard DHCPv6 options, so a separate option space must be defined.

Note that the creation of a new option space is not required when defining sub-options for a standard option, because it is created by default if the standard option is meant to convey any sub-options (see Section 9.2.14).

Assume that we want to have a DHCPv6 option called "container" with code 102 that conveys two sub-options with codes 1 and 2. First we need to define the new sub-options:

```
"Dhcp6": {  
  "option-def": [  
    {  
      "name": "subopt1",  
      "code": 1,  
      "space": "isc",  
      "type": "ipv6-address",  
      "record-types": "",  
      "array": false,  
      "encapsulate": ""  
    },  
    {  
      "name": "subopt2",  
      "code": 2,  
      "space": "isc",  
      "type": "string",  
      "record-types": "",  
      "array": false,  
      "encapsulate": ""  
    }  
  ],  
  ...  
}
```

Note that we have defined the options to belong to a new option space (in this case, "isc").

The next step is to define a regular DHCPv6 option and specify that it should include options from the new option space:



```
"Dhcp6": {
  "option-def": [
    ...,
    {
      "name": "container",
      "code": 102,
      "space": "dhcp6",
      "type": "empty",
      "array": false,
      "record-types": "",
      "encapsulate": "isc"
    }
  ],
  ...
}
```

The name of the option space in which the sub-options are defined is set in the **encapsulate** field. The **type** field is set to **empty**, which limits this option to only carrying data in sub-options.

Finally, we can set values for the new options:

```
"Dhcp6": {
  "option-data": [
    {
      "name": "subopt1",
      "code": 1,
      "space": "isc",
      "data": "2001:db8::abcd"
    },
    {
      "name": "subopt2",
      "code": 2,
      "space": "isc",
      "data": "Hello world"
    },
    {
      "name": "container",
      "code": 102,
      "space": "dhcp6"
    }
  ],
  ...
}
```

Note that it is possible to create an option which carries some data in addition to the sub-options defined in the encapsulated option space. For example, if the "container" option from the previous example were required to carry a uint16 value as well as the sub-options, the **type** value would have to be set to "uint16" in the option definition. (Such an option would then have the following data structure: DHCP header, uint16 value, sub-options.) The value specified with the **data** parameter — which should be a valid integer enclosed in quotes, e.g. "123" — would then be assigned to the uint16 field in the "container" option.

9.2.16 Unspecified Parameters for DHCPv6 Option Configuration

In many cases it is not required to specify all parameters for an option configuration and the default values can be used. However, it is important to understand the implications of not specifying some of them, as it may result in configuration errors. The list below explains the behavior of the server when a particular parameter is not explicitly specified:

- **name** - the server requires an option name or option code to identify an option. If this parameter is unspecified, the option code must be specified.



- **code** - the server requires an option name or option code to identify an option. This parameter may be left unspecified if the **name** parameter is specified. However, this also requires that the particular option have a definition (either as a standard option or an administrator-created definition for the option using an 'option-def' structure), as the option definition associates an option with a particular name. It is possible to configure an option for which there is no definition (unspecified option format). Configuration of such options requires the use of the option code.
- **space** - if the option space is unspecified it will default to 'dhcp6', which is an option space holding standard DHCPv6 options.
- **data** - if the option data is unspecified it defaults to an empty value. The empty value is mostly used for the options which have no payload (boolean options), but it is legal to specify empty values for some options which carry variable-length data and which the specification allows a length of 0. For such options, the data parameter may be omitted in the configuration.
- **csv-format** - if this value is not specified the server will assume that the option data is specified as a list of comma-separated values to be assigned to individual fields of the DHCP option.

9.2.17 Controlling the Values Sent for T1 and T2 Times

According to [RFC 8415, section 21.4](#), the recommended T1 and T2 values are 50% and 80% of the preferred lease time, respectively. Kea can be configured to send values that are specified explicitly or that are calculated as percentages of the preferred lease time. The server's behavior is governed by combination of configuration parameters, two of which have already been mentioned.

Beginning with Kea 1.6.0 lease preferred and valid lifetime are extended from single values to triplets with minimum, default and maximum values using:

- **min-preferred-lifetime** - specifies the minimum preferred lifetime (optional).
- **preferred-lifetime** - specifies the default preferred lifetime.
- **max-preferred-lifetime** - specifies the maximum preferred lifetime (optional).
- **min-valid-lifetime** - specifies the minimum valid lifetime (optional).
- **valid-lifetime** - specifies the default valid lifetime.
- **max-valid-lifetime** - specifies the maximum valid lifetime (optional).

When the client does not specify lifetimes the default is used. When it specifies a lifetime using IAADDR or IAPREFIX sub option with not zero values these values are used when they are between configured minimum (lower values are round up) and maximum (larger values are round down) bounds.

To send specific, fixed values use the following two parameters:

- **renew-timer** - specifies the value of T1 in seconds.
- **rebind-timer** - specifies the value of T2 in seconds.

You may specify any value for T2 greater than or equal to zero. When specifying T1 it must be less than T2. This flexibility is allowed to support a use case where admins want to suppress client renewals and rebinds by deferring them beyond the life span of the lease. This should cause the lease to expire, rather than get renewed by clients. If T1 is specified as larger than T2, it will be set to zero in the outbound IA.

In great majority of cases the values should follow this rule: $T1 < T2 < \text{preferred lifetime} < \text{valid lifetime}$. Alternatively, both T1 and T2 values can be configured to 0, which is a signal to DHCPv6 clients that they may renew at their own discretion. However, there are known broken client implementations out there that will start renewing immediately. If you plan to use $T1=T2=0$ values, please test first and make sure your clients behave rationally.

In some rare cases there may be a need to disable client's ability to renew addresses. This is undesired from protocol perspective and should be avoided if possible. However, if you want to do this, you can configure your T1 and T2 values to be equal or greater to your valid lifetime. Be advised that this will cause your clients to occasionally lose their addresses, which is generally perceived as poor service. However, there may be some rare business cases when this is desired (e.g. when you want to break long lasting connections on purpose).

Calculating the values is controlled by the following three parameters.



- **calculate-tee-times** - when true, T1 and T2 will be calculated as percentages of the valid lease time. It defaults to true.
- **t1-percent** - the percentage of the valid lease time to use for T1. It is expressed as a real number between 0.0 and 1.0 and must be less than t2-percent. The default value is 0.5 per RFC 8415.
- **t2-percent** - the percentage of the valid lease time to use for T2. It is expressed as a real number between 0.0 and 1.0 and must be greater than t1-percent. The default value is 0.8 per RFC 8415.

Note In the event that both explicit values are specified and calculate-tee-times is true, the server will use the explicit values. If you plan on having a mixture where some subnets or share-networks will use explicit values and some will use calculated values you must not define the explicit values at any level higher than where they will be used. Inheriting them from too high of a scope, such as global, will cause them to have values at every level underneath (shared-networks and subnets), effectively disabling calculated values.

9.2.18 IPv6 Subnet Selection

The DHCPv6 server may receive requests from local (connected to the same subnet as the server) and remote (connected via relays) clients. As the server may have many subnet configurations defined, it must select an appropriate subnet for a given request.

In IPv4, the server can determine which of the configured subnets are local, as there is a reasonable expectation that the server will have a (global) IPv4 address configured on the interface, and it can use that information to detect whether a subnet is local. That assumption is not true in IPv6; the DHCPv6 server must be able to operate while only using link-local addresses. Therefore, an optional **interface** parameter is available within a subnet definition to designate that a given subnet is local, i.e. reachable directly over the specified interface. For example, the server that is intended to serve a local subnet over eth0 may be configured as follows:

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:beef::/48",
      "pools": [
        {
          "pool": "2001:db8:beef::/48"
        }
      ],
      "interface": "eth0"
    }
  ],
  ...
}
```

9.2.19 Rapid Commit

The Rapid Commit option, described in [RFC 8415](#), is supported by the Kea DHCPv6 server. However, support is disabled by default. It can be enabled on a per-subnet basis using the **rapid-commit** parameter as shown below:

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:beef::/48",
      "rapid-commit": true,
      "pools": [
        {
          "pool": "2001:db8:beef::1-2001:db8:beef::10"
        }
      ],
    }
  ],
}
```



```
    }  
  ],  
  ...  
}
```

This setting only affects the subnet for which the **rapid-commit** is set to **true**. For clients connected to other subnets, the server will ignore the Rapid Commit option sent by the client and will follow the 4-way exchange procedure, i.e. respond with an Advertise for a Solicit containing a Rapid Commit option.

9.2.20 DHCPv6 Relays

A DHCPv6 server with multiple subnets defined must select the appropriate subnet when it receives a request from a client. For clients connected via relays, two mechanisms are used:

The first uses the linkaddr field in the RELAY_FORW message. The name of this field is somewhat misleading in that it does not contain a link-layer address; instead, it holds an address (typically a global address) that is used to identify a link. The DHCPv6 server checks to see whether the address belongs to a defined subnet and, if it does, that subnet is selected for the client's request.

The second mechanism is based on interface-id options. While forwarding a client's message, relays may insert an interface-id option into the message that identifies the interface on the relay that received the message. (Some relays allow configuration of that parameter, but it is sometimes hardcoded and may range from the very simple (e.g. "vlan100") to the very cryptic; one example seen on real hardware was "ISAM144|299|ip6|nt:vp:1:110"). The server can use this information to select the appropriate subnet. The information is also returned to the relay, which then knows the interface to use to transmit the response to the client. For this to work successfully, the relay interface IDs must be unique within the network and the server configuration must match those values.

When configuring the DHCPv6 server, it should be noted that two similarly named parameters can be configured for a subnet:

- **interface** defines which local network interface can be used to access a given subnet.
- **interface-id** specifies the content of the interface-id option used by relays to identify the interface on the relay to which the response packet is sent.

The two are mutually exclusive; a subnet cannot be reachable both locally (direct traffic) and via relays (remote traffic). Specifying both is a configuration error and the DHCPv6 server will refuse such a configuration.

The following example configuration shows how to specify an interface-id with a value of "vlan123":

```
"Dhcp6": {  
  "subnet6": [  
    {  
      "subnet": "2001:db8:beef::/48",  
      "pools": [  
        {  
          "pool": "2001:db8:beef::/48"  
        }  
      ],  
      "interface-id": "vlan123"  
    }  
  ],  
  ...  
}
```

9.2.21 Relay-Supplied Options

[RFC 6422](#) defines a mechanism called Relay-Supplied DHCP Options. In certain cases relay agents are the only entities that may have specific information, and they can insert options when relaying messages from the client to the server. The server will then do certain checks and copy those options to the response sent to the client.



There are certain conditions that must be met for the option to be included. First, the server must not provide the option itself; in other words, if both relay and server provide an option, the server always takes precedence. Second, the option must be RSOO-enabled. (RSOO is the "Relay Supplied Options option.") IANA maintains a list of RSOO-enabled options [here](#). However, there may be cases when system administrators want to echo other options. Kea can be instructed to treat other options as RSOO-enabled. For example, to mark options 110, 120, and 130 as RSOO-enabled, the following syntax should be used:

```
"Dhcp6": {  
  "relay-supplied-options": [ "110", "120", "130" ],  
  ...  
}
```

As of February 2019, only option 65 is RSOO-enabled by IANA. This option will always be treated as such, so there is no need to explicitly mark it. Also, when enabling standard options, it is possible to use their names, rather than option code, e.g. use **dns-servers** instead of **23**. See Table 9.1 for the names. In certain cases it could also work for custom options, but due to the nature of the parser code this may be unreliable and should be avoided.

9.2.22 Client Classification in DHCPv6

The DHCPv6 server includes support for client classification. For a deeper discussion of the classification process see Chapter 14.

In certain cases it is useful to configure the server to differentiate between DHCP client types and treat them accordingly. Client classification can be used to modify the behavior of almost any part of the DHCP message processing. In the current release of Kea, there are three mechanisms that take advantage of client classification in DHCPv6: subnet selection, address pool selection, and DHCP options assignment.

Kea can be instructed to limit access to given subnets based on class information. This is particularly useful for cases where two types of devices share the same link and are expected to be served from two different subnets. The primary use case for such a scenario is cable networks, where there are two classes of devices: the cable modem itself, which should be handed a lease from subnet A; and all other devices behind the modem, which should get a lease from subnet B. That segregation is essential to prevent overly curious users from playing with their cable modems. For details on how to set up class restrictions on subnets, see Section 14.6.

When subnets belong to a shared network, the classification applies to subnet selection but not to pools, e.g., a pool in a subnet limited to a particular class can still be used by clients which do not belong to the class, if the pool they are expected to use is exhausted. So the limit on access based on class information is also available at the address/prefix pool level; see Section 14.7, within a subnet. This is useful when segregating clients belonging to the same subnet into different address ranges.

In a similar way, a pool can be constrained to serve only known clients, i.e. clients which have a reservation, using the built-in "KNOWN" or "UNKNOWN" classes. One can assign addresses to registered clients without giving a different address per reservation, for instance when there are not enough available addresses. The determination whether there is a reservation for a given client is made after a subnet is selected, so it is not possible to use KNOWN/UNKNOWN classes to select a shared network or a subnet.

The process of classification is conducted in five steps. The first step is to assess an incoming packet and assign it to zero or more classes. Next, a subnet is chosen, possibly based on the class information. When the incoming packet is in the special class, "DROP", it is dropped and a debug message logged. After that, class expressions are evaluated depending on the built-in "KNOWN"/"UNKNOWN" classes after host reservation lookup, using them for pool/pd-pool selection and assigning classes from host reservations. The list of required classes is then built and each class of the list has its expression evaluated; when it returns "true" the packet is added as a member of the class. Finally, options are assigned, again possibly based on the class information. More complete and detailed information is available in Chapter 14.

There are two main methods of classification. The first is automatic and relies on examining the values in the vendor class options or the existence of a host reservation. Information from these options is extracted, and a class name is constructed from it and added to the class list for the packet. The second specifies an expression that is evaluated for each packet. If the result is "true", the packet is a member of the class.

Note

Care should be taken with client classification, as it is easy for clients that do not meet class criteria to be denied all service.



9.2.22.1 Defining and Using Custom Classes

The following example shows how to configure a class using an expression and a subnet using that class. This configuration defines the class named "Client_enterprise". It is comprised of all clients whose client identifiers start with the given hex string (which would indicate a DUID based on an enterprise id of 0xAABBCCDD). They will be given an address from 2001:db8:1::0 to 2001:db8:1::FFFF and the addresses of their DNS servers set to 2001:db8:0::1 and 2001:db8:2::1.

```
"Dhcp6": {
  "client-classes": [
    {
      "name": "Client_enterprise",
      "test": "substring(option[1].hex,0,6) == 0x0002AABBCCDD",
      "option-data": [
        {
          "name": "dns-servers",
          "code": 23,
          "space": "dhcp6",
          "csv-format": true,
          "data": "2001:db8:0::1, 2001:db8:2::1"
        }
      ]
    }
  ],
  ...
},
"subnet6": [
  {
    "subnet": "2001:db8:1::/64",
    "pools": [ { "pool": "2001:db8:1::-2001:db8:1::ffff" } ],
    "client-class": "Client_enterprise"
  }
],
...
}
```

This example shows a configuration using an automatically generated "VENDOR_CLASS_" class. The administrator of the network has decided that addresses in the range 2001:db8:1::1 to 2001:db8:1::ffff are to be managed by the DHCP6 server and that only clients belonging to the eRouter1.0 client class are allowed to use that pool.

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "pools": [
        {
          "pool": "2001:db8:1::-2001:db8:1::ffff"
        }
      ]
    },
    "client-class": "VENDOR_CLASS_eRouter1.0"
  ]
},
...
}
```

9.2.22.2 Required Classification

In some cases it is useful to limit the scope of a class to a shared-network, subnet, or pool. There are two parameters which are used to limit the scope of the class by instructing the server to perform evaluation of test expressions when required.

The first one is the per-class **only-if-required** flag which is false by default. When it is set to **true**, the test expression of the class is not evaluated at the reception of the incoming packet but later, and only if the class evaluation is required.



The second is **require-client-classes**, which takes a list of class names and is valid in shared-network, subnet, and pool scope. Classes in these lists are marked as required and evaluated after selection of this specific shared-network/subnet/pool and before output option processing.

In this example, a class is assigned to the incoming packet when the specified subnet is used:

```
"Dhcp6": {
  "client-classes": [
    {
      "name": "Client_foo",
      "test": "member('ALL')",
      "only-if-required": true
    },
    ...
  ],
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64"
      "pools": [
        {
          "pool": "2001:db8:1::-2001:db8:1::ffff"
        }
      ],
      "require-client-classes": [ "Client_foo" ],
      ...
    },
    ...
  ],
  ...
}
```

Required evaluation can be used to express complex dependencies, for example, subnet membership. It can also be used to reverse the precedence; if you set an option-data in a subnet it takes precedence over an option-data in a class. When you move the option-data to a required class and require it in the subnet, a class evaluated earlier may take precedence.

Required evaluation is also available at shared-network and pool/pd-pool levels. The order in which required classes are considered is: shared-network, subnet, and (pd-)pool, i.e. the opposite order that option-data is processed.

9.2.23 DDNS for DHCPv6

As mentioned earlier, kea-dhcp6 can be configured to generate requests to the DHCP-DDNS server (referred to here as "D2") to update DNS entries. These requests are known as Name Change Requests or NCRs. Each NCR contains the following information:

1. Whether it is a request to add (update) or remove DNS entries
2. Whether the change requests forward DNS updates (AAAA records), reverse DNS updates (PTR records), or both
3. The Fully Qualified Domain Name (FQDN), lease address, and DHCID (information identifying the client associated with the FQDN)

The parameters controlling the generation of NCRs for submission to D2 are contained in the **dhcp-ddns** section of the kea-dhcp6 server configuration. The mandatory parameters for the DHCP DDNS configuration are **enable-updates**, which is unconditionally required, and **qualifying-suffix**, which has no default value and is required when **enable-updates** is set to **true**. The two (disabled and enabled) minimal DHCP DDNS configurations are:

```
"Dhcp6": {
  "dhcp-ddns": {
    "enable-updates": false
  },
  ...
}
```



and for example:

```
"Dhcp6": {
  "dhcp-ddns": {
    "enable-updates": true,
    "qualifying-suffix": "example."
  },
  ...
}
```

The default values for the "dhcp-ddns" section are as follows:

- "server-ip": "127.0.0.1"
- "server-port": 53001
- "sender-ip": ""
- "sender-port": 0
- "max-queue-size": 1024
- "ncr-protocol": "UDP"
- "ncr-format": "JSON"
- "override-no-update": false
- "override-client-update": false
- "replace-client-name": "never"
- "generated-prefix": "myhost"
- "hostname-char-set": ""
- "hostname-char-replacement": ""

9.2.23.1 DHCP-DDNS Server Connectivity

For NCRs to reach the D2 server, kea-dhcp6 must be able to communicate with it. kea-dhcp6 uses the following configuration parameters to control this communication:

- **enable-updates** - determines whether kea-dhcp6 will generate NCRs. If missing, this value is assumed to be false, so DDNS updates are disabled. To enable DDNS updates set this value to true.
- **server-ip** - IP address on which D2 listens for requests. The default is the local loopback interface at address 127.0.0.1. You may specify either an IPv4 or IPv6 address.
- **server-port** - port on which D2 listens for requests. The default value is 53001.
- **sender-ip** - IP address which kea-dhcp6 uses to send requests to D2. The default value is blank, which instructs kea-dhcp6 to select a suitable address.
- **sender-port** - port which kea-dhcp6 uses to send requests to D2. The default value of 0 instructs kea-dhcp6 to select a suitable port.
- **max-queue-size** - maximum number of requests allowed to queue waiting to be sent to D2. This value guards against requests accumulating uncontrollably if they are being generated faster than they can be delivered. If the number of requests queued for transmission reaches this value, DDNS updating will be turned off until the queue backlog has been sufficiently reduced. The intent is to allow kea-dhcp6 to continue lease operations without running the risk that its memory usage grows without limit. The default value is 1024.



- **ncr-protocol** - socket protocol to use when sending requests to D2. Currently only UDP is supported.
- **ncr-format** - packet format to use when sending requests to D2. Currently only JSON format is supported.

By default, kea-dhcp-ddns is assumed to be running on the same machine as kea-dhcp6, and all of the default values mentioned above should be sufficient. If, however, D2 has been configured to listen on a different address or port, these values must be altered accordingly. For example, if D2 has been configured to listen on 2001:db8::5 port 900, the following configuration is required:

```
"Dhcp6": {
  "dhcp-ddns": {
    "server-ip": "2001:db8::5",
    "server-port": 900,
    ...
  },
  ...
}
```

9.2.23.2 When Does the kea-dhcp6 Server Generate a DDNS Request?

kea-dhcp6 follows the behavior prescribed for DHCP servers in [RFC 4704](#). It is important to keep in mind that kea-dhcp6 makes the initial decision of when and what to update and forwards that information to D2 in the form of NCRs. Carrying out the actual DNS updates and dealing with such things as conflict resolution are within the purview of D2 itself (Chapter 12). This section describes when kea-dhcp6 will generate NCRs and the configuration parameters that can be used to influence this decision. It assumes that the **enable-updates** parameter is true.

Note

Currently the interface between kea-dhcp6 and D2 only supports requests which update DNS entries for a single IP address. If a lease grants more than one address, kea-dhcp6 will create the DDNS update request for only the first of these addresses.

In general, kea-dhcp6 will generate DDNS update requests when:

1. A new lease is granted in response to a DHCPREQUEST
2. An existing lease is renewed but the FQDN associated with it has changed
3. An existing lease is released in response to a DHCPRELEASE

In the second case, lease renewal, two DDNS requests will be issued: one request to remove entries for the previous FQDN, and a second request to add entries for the new FQDN. In the last case, a lease release, a single DDNS request to remove its entries will be made.

The decisions involved when granting a new lease (the first case) are more complex. When a new lease is granted, kea-dhcp6 will generate a DDNS update request only if the DHCPREQUEST contains the FQDN option (code 39). By default, kea-dhcp6 will respect the FQDN N and S flags specified by the client as shown in the following table:

The first row in the table above represents "client delegation". Here the DHCP client states that it intends to do the forward DNS updates and the server should do the reverse updates. By default, kea-dhcp6 will honor the client's wishes and generate a DDNS request to D2 to update only reverse DNS data. The parameter **override-client-update** can be used to instruct the server to override client delegation requests. When this parameter is true, kea-dhcp6 will disregard requests for client delegation and generate a DDNS request to update both forward and reverse DNS data. In this case, the N-S-O flags in the server's response to the client will be 0-1-1 respectively.

(Note that the flag combination N=1, S=1 is prohibited according to [RFC 4702](#). If such a combination is received from the client, the packet will be dropped by kea-dhcp6.)

To override client delegation, set the following values in the configuration file:



Client Flags:N-S	Client Intent	Server Response	Server Flags:N-S-O
0-0	Client wants to do forward updates, server should do reverse updates	Server generates reverse-only request	1-0-0
0-1	Server should do both forward and reverse updates	Server generates request to update both directions	0-1-0
1-0	Client wants no updates done	Server does not generate a request	1-0-0

Table 9.2: Default FQDN Flag Behavior

```
"Dhcp6": {
  "dhcp-ddns": {
    "override-client-update": true,
    ...
  },
  ...
}
```

The third row in the table above describes the case in which the client requests that no DNS updates be done. The parameter, **override-no-update**, can be used to instruct the server to disregard the client's wishes. When this parameter is true, kea-dhcp6 will generate DDNS update requests to kea-dhcp-ddns even if the client requests no updates be done. The N-S-O flags in the server's response to the client will be 0-1-1.

To override client delegation, issue the following commands:

```
"Dhcp6": {
  "dhcp-ddns": {
    "override-no-update": true,
    ...
  },
  ...
}
```

9.2.23.3 kea-dhcp6 Name Generation for DDNS Update Requests

Each Name Change Request must of course include the fully qualified domain name whose DNS entries are to be affected. kea-dhcp6 can be configured to supply a portion or all of that name, based upon what it receives from the client in the DHCPREQUEST.

The default rules for constructing the FQDN that will be used for DNS entries are:

1. If the DHCPREQUEST contains the client FQDN option, take the candidate name from there.
2. If the candidate name is a partial (i.e. unqualified) name, then add a configurable suffix to the name and use the result as the FQDN.
3. If the candidate name provided is empty, generate an FQDN using a configurable prefix and suffix.
4. If the client provided neither option, then no DNS action will be taken.

These rules can be amended by setting the **replace-client-name** parameter, which provides the following modes of behavior:

- **never** - Use the name the client sent. If the client sent no name, do not generate one. This is the default mode.
- **always** - Replace the name the client sent. If the client sent no name, generate one for the client.
- **when-present** - Replace the name the client sent. If the client sent no name, do not generate one.



- **when-not-present** - Use the name the client sent. If the client sent no name, generate one for the client.

Note Note that in early versions of Kea, this parameter was a boolean and permitted only values of **true** and **false**. Boolean values have been deprecated and are no longer accepted. If you are currently using booleans, you must replace them with the desired mode name. A value of **true** maps to "**when-present**", while **false** maps to "**never**".

For example, to instruct kea-dhcp6 to always generate the FQDN for a client, set the parameter **replace-client-name** to **always** as follows:

```
"Dhcp6": {
  "dhcp-ddns": {
    "replace-client-name": "always",
    ...
  },
  ...
}
```

The prefix used in the generation of an FQDN is specified by the **generated-prefix** parameter. The default value is "myhost". To alter its value, simply set it to the desired string:

```
"Dhcp6": {
  "dhcp-ddns": {
    "generated-prefix": "another.host",
    ...
  },
  ...
}
```

The suffix used when generating an FQDN, or when qualifying a partial name, is specified by the **qualifying-suffix** parameter. This parameter has no default value, thus it is mandatory when DDNS updates are enabled. To set its value simply set it to the desired string:

```
"Dhcp6": {
  "dhcp-ddns": {
    "qualifying-suffix": "foo.example.org",
    ...
  },
  ...
}
```

When qualifying a partial name, kea-dhcp6 will construct the name in the format:

[candidate-name].[qualifying-suffix].

where candidate-name is the partial name supplied in the DHCPREQUEST. For example, if the FQDN domain name value is "some-computer" and the qualifying-suffix "example.com", the generated FQDN is:

some-computer.example.com.

When generating the entire name, kea-dhcp6 will construct the name in the format:

[generated-prefix]-[address-text].[qualifying-suffix].

where address-text is simply the lease IP address converted to a hyphenated string. For example, if the lease address is 3001:1::70E, the qualifying suffix "example.com", and the default value is used for **generated-prefix**, the generated FQDN would be:

myhost-3001-1--70E.example.com.



9.2.23.4 Sanitizing Client FQDN Names

It may be that some of your DHCP clients provide values in the name component of the FQDN option (Option code 39) that contain undesirable characters. It is possible to configure kea-dhcp6 to sanitize these values. The most typical use case is ensuring that only characters that are permitted by RFC 1035 be included: A-Z,a-z,0-9, and '-'. This may be accomplished with the following two parameters:

- **hostname-char-set** - a regular expression describing the invalid character set. This can be any valid, regular expression using POSIX extended expression syntax. For example, "[^A-Za-z0-9-]" would replace any character other than the letters A through z, digits 0 through 9, and '-'. An empty string, the default value, disables sanitization.
- **hostname-char-replacement** - a string of zero or more characters with which to replace each invalid character in the client value. The default value is an empty string and will cause invalid characters to be OMITTED rather than replaced.

The following configuration will replace anything other than a letter, digit, hyphen, or dot with the letter 'x':

```
"Dhcp4": {
  "dhcp-ddns": {
    "hostname-char-set": "[^A-Za-z0-9-]",
    "hostname-char-replacement": "x",
    ...
  },
  ...
}
```

Thus, a client supplied value of "myhost-\$(123.org)" would become "myhost-xx123.org". Sanitizing is performed only on the portion of the name supplied by the client, and it is performed before applying a qualifying suffix (if one is defined and needed).

Note The following are some considerations to keep in mind:

Name sanitizing is meant to catch the more common cases of invalid characters through a relatively simple character-replacement scheme. It is difficult to devise a scheme that works well in all cases. If you find you have clients that are using odd corner cases of character combinations that cannot be readily handled with this mechanism, you should consider writing a hook that can carry out sufficiently complex logic to address your needs.

Do not include dots in your hostname-char-set expression. When scrubbing FQDNs, dots are treated as delimiters and used to separate the option value into individual domain labels that are scrubbed and then re-assembled.

If your clients are sending values that differ only by characters considered as invalid by your hostname-char-set, be aware that scrubbing them will yield identical values. In such cases, DDNS conflict rules will permit only one of them to register the name. Finally, given the latitude clients have in the values they send, it is virtually impossible to guarantee that a combination of these two parameters will always yield a name that is valid for use in DNS. For example, using an empty value for hostname-char-replacement could yield an empty domain label within a name, if that label consists only of invalid characters.

Note Since the 1.6.0 Kea release it is possible to specify hostname-char-set and/or hostname-char-replacement at the global scope. This allows to sanitize host names without requiring a dhcp-ddns entry. When a hostname-char parameter is defined at the global scope and in a dhcp-ddns entry the second (local) value is used.

9.2.24 DHCPv4-over-DHCPv6: DHCPv6 Side

The support of DHCPv4-over-DHCPv6 transport is described in [RFC 7341](#) and is implemented using cooperating DHCPv4 and DHCPv6 servers. This section is about the configuration of the DHCPv6 side (the DHCPv4 side is described in [Section 8.2.23](#)).

Note DHCPv4-over-DHCPv6 support is experimental and the details of the inter-process communication may change; both the DHCPv4 and DHCPv6 sides should be running the same version of Kea. For instance, the support of port relay (RFC 8357) introduced an incompatible change.



There is only one specific parameter for the DHCPv6 side: **dhcp4o6-port**, which specifies the first of the two consecutive ports of the UDP sockets used for the communication between the DHCPv6 and DHCPv4 servers (the DHCPv6 server is bound to `::1` on **port** and connected to `::1` on **port + 1**).

Two other configuration entries are generally required: unicast traffic support (see Section 9.2.7) and DHCP 4o6 server address option (name "dhcp4o6-server-addr", code 88).

The following configuration was used during some tests:

```
{
# DHCPv6 conf
"Dhcp6": {

  "interfaces-config": {
    "interfaces": [ "eno33554984/2001:db8:1:1::1" ]
  },

  "lease-database": {
    "type": "memfile",
    "name": "leases6"
  },

  "preferred-lifetime": 3000,
  "valid-lifetime": 4000,
  "renew-timer": 1000,
  "rebind-timer": 2000,

  "subnet6": [ {
    "subnet": "2001:db8:1:1::/64",
    "interface": "eno33554984",
    "pools": [ { "pool1": "2001:db8:1:1::1:0/112" } ]
  } ],

  "dhcp4o6-port": 6767,

  "option-data": [ {
    "name": "dhcp4o6-server-addr",
    "code": 88,
    "space": "dhcp6",
    "csv-format": true,
    "data": "2001:db8:1:1::1"
  } ],

  "loggers": [ {
    "name": "kea-dhcp6",
    "output_options": [ {
      "output": "/tmp/kea-dhcp6.log"
    } ],
    "severity": "DEBUG",
    "debuglevel": 0
  } ]
}
}
```

Note Relayed DHCPv4-QUERY DHCPv6 messages are not supported.



9.2.25 Sanity Checks in DHCPv6

An important aspect of a well-running DHCP system is an assurance that the data remains consistent. However, in some cases it may be convenient to tolerate certain inconsistent data. For example, a network administrator that temporarily removed a subnet from a configuration wouldn't want all the leases associated with it disappear from the lease database. Kea has a mechanism to control sanity checks for situations such as this.

Kea supports a configuration scope called **sanity-checks**. It currently allows only a single parameter called **lease-checks**, which governs the verification carried out when a new lease is loaded from a lease file. This mechanism permits Kea to attempt to correct inconsistent data.

Every subnet has a subnet-id value; this is how Kea internally identifies subnets. Each lease has a subnet-id parameter as well, which identifies which subnet it belongs to. However, if the configuration has changed, it is possible that a lease could exist with a subnet-id, but without any subnet that matches it. Also, it may be possible that the subnet's configuration has changed and the subnet-id now belongs to a subnet that does not match the lease. Kea's corrective algorithm first checks to see if there is a subnet with the subnet-id specified by the lease. If there is, it verifies whether the lease belongs to that subnet. If not, depending on the lease-checks setting, the lease is discarded, a warning is displayed, or a new subnet is selected for the lease that matches it topologically.

Since delegated prefixes do not have to belong to a subnet in which they're offered, there is no way to implement such a mechanism for IPv6 prefixes. As such, the mechanism works for IPv6 addresses only.

There are five levels which are supported:

- **none** - do no special checks; accept the lease as is
- **warn** - if problems are detected, a warning will be displayed, but the lease data will be accepted anyway. This is the default value.
- **fix** - If data inconsistency is discovered, Kea will try to correct it. If the correction is not successful, the incorrect data will be inserted anyway.
- **fix-del** - If a data inconsistency is discovered, Kea will try to correct it. If the correction is not successful, the lease will be rejected. This setting ensures the data's correctness, but some incorrect data may be lost. Use with care.
- **del** - This is the strictest mode. If any inconsistency is detected, the lease is rejected. Use with care.

This feature is currently implemented for the memfile backend.

An example configuration that sets this parameter looks as follows:

```
"Dhcp6": {
  "sanity-checks": {
    "lease-checks": "fix-del"
  },
  ...
}
```

9.3 Host Reservation in DHCPv6

There are many cases where it is useful to provide a configuration on a per-host basis. The most obvious one is to reserve a specific, static IPv6 address or/and prefix for exclusive use by a given client (host); the returning client will get the same address or/and prefix every time, and other clients will never get that address. Another example when host reservations are applicable is when a host has specific requirements, e.g. a printer that needs additional DHCP options or a cable modem that needs specific parameters. Yet another possible use case is to define unique names for hosts.

Note that there may be cases when a new reservation has been made for a client for the address or prefix currently in use by another client. We call this situation a "conflict." The conflicts get resolved automatically over time as described in subsequent sections. Once the conflict is resolved, the client will keep receiving the reserved configuration when it renews.



Host reservations are defined as parameters for each subnet. Each host must be identified by either DUID or its hardware/MAC address. See Section 9.11 for details. There is an optional **reservations** array in the **subnet6** structure. Each element in that array is a structure that holds information about a single host. In particular, the structure has an identifier that uniquely identifies a host. In the DHCPv6 context, such an identifier is usually a DUID, but can also be a hardware or MAC address. One or more addresses or prefixes may also be specified, and it is possible to specify a hostname and DHCPv6 options for a given host.

The following example shows how to reserve addresses and prefixes for specific hosts:

```
"subnet6": [
  {
    "subnet": "2001:db8:1::/48",
    "pools": [ { "pool": "2001:db8:1::/80" } ],
    "pd-pools": [
      {
        "prefix": "2001:db8:1:8000::",
        "prefix-len": 48,
        "delegated-len": 64
      }
    ],
    "reservations": [
      {
        "duid": "01:02:03:04:05:0A:0B:0C:0D:0E",
        "ip-addresses": [ "2001:db8:1::100" ]
      },
      {
        "hw-address": "00:01:02:03:04:05",
        "ip-addresses": [ "2001:db8:1::101", "2001:db8:1::102" ]
      },
      {
        "duid": "01:02:03:04:05:06:07:08:09:0A",
        "ip-addresses": [ "2001:db8:1::103" ],
        "prefixes": [ "2001:db8:2:abcd::/64" ],
        "hostname": "foo.example.com"
      }
    ]
  }
]
```

This example includes reservations for three different clients. The first reservation is for the address 2001:db8:1::100 for a client using DUID 01:02:03:04:05:0A:0B:0C:0D:0E. The second reservation is for two addresses 2001:db8:1::101 and 2001:db8:1::102, for a client using MAC address 00:01:02:03:04:05. Lastly, address 2001:db8:1::103 and prefix 2001:db8:2:abcd::/64 are reserved for a client using DUID 01:02:03:04:05:06:07:08:09:0A. The last reservation also assigns a hostname to this client.

Note that DHCPv6 allows a single client to lease multiple addresses and multiple prefixes at the same time. Therefore **ip-addresses** and **prefixes** are plural and are actually arrays. When the client sends multiple IA options (IA_NA or IA_PD), each reserved address or prefix is assigned to an individual IA of the appropriate type. If the number of IAs of a specific type is lower than the number of reservations of that type, the number of reserved addresses or prefixes assigned to the client is equal to the number of IA_NAs or IA_PDAs sent by the client; that is, some reserved addresses or prefixes are not assigned. However, they still remain reserved for this client and the server will not assign them to any other client. If the number of IAs of a specific type sent by the client is greater than the number of reserved addresses or prefixes, the server will try to assign all reserved addresses or prefixes to the individual IAs and dynamically allocate addresses or prefixes to the remaining IAs. If the server cannot assign a reserved address or prefix because it is in use, the server will select the next reserved address or prefix and try to assign it to the client. If the server subsequently finds that there are no more reservations that can be assigned to the client at that moment, the server will try to assign leases dynamically.

Making a reservation for a mobile host that may visit multiple subnets requires a separate host definition in each subnet it is expected to visit. It is not possible to define multiple host definitions with the same hardware address in a single subnet. Multiple host definitions with the same hardware address are valid if each is in a different subnet. The reservation for a given host should include only one identifier, either DUID or hardware address. Defining both for the same host is considered a configuration error.

Adding host reservations incurs a performance penalty. In principle, when a server that does not support host reservation responds to a query, it needs to check whether there is a lease for a given address being considered for allocation or renewal. The server



that also supports host reservation has to perform additional checks: not only whether the address is currently used (i.e., if there is a lease for it), but also whether the address could be used by someone else (i.e., if there is a reservation for it). That additional check incurs extra overhead.

9.3.1 Address/Prefix Reservation Types

In a typical scenario there is an IPv6 subnet defined, with a certain part of it dedicated for dynamic address allocation by the DHCPv6 server. There may be an additional address space defined for prefix delegation. Those dynamic parts are referred to as dynamic pools, address and prefix pools, or simply pools. In principle, a host reservation can reserve any address or prefix that belongs to the subnet. The reservations that specify addresses that belongs to configured pools are called "in-pool reservations." In contrast, those that do not belong to dynamic pools are called "out-of-pool reservations." There is no formal difference in the reservation syntax and both reservation types are handled uniformly.

Kea supports global host reservations. These are reservations that are specified at the global level within the configuration and that do not belong to any specific subnet. Kea will still match inbound client packets to a subnet as before, but when the subnet's reservation mode is set to "**global**", Kea will look for host reservations only among the global reservations defined. Typically, such reservations would be used to reserve hostnames for clients which may move from one subnet to another.

Note You can reserve any ip-address or prefix in a global reservation. Just keep in mind that Kea will not do any sanity checking on the address or prefix and that for Kea 1.5.0, support for global reservations should be considered experimental.

9.3.2 Conflicts in DHCPv6 Reservations

As reservations and lease information are stored separately, conflicts may arise. Consider the following series of events: the server has configured the dynamic pool of addresses from the range of 2001:db8::10 to 2001:db8::20. Host A requests an address and gets 2001:db8::10. Now the system administrator decides to reserve address 2001:db8::10 for Host B. In general, reserving an address that is currently assigned to someone else is not recommended, but there are valid use cases where such an operation is warranted.

The server now has a conflict to resolve. If Host B boots up and requests an address, the server is not able to assign the reserved address 2001:db8::10. A naive approach would be to immediately remove the lease for Host A and create a new one for Host B. That would not solve the problem, though, because as soon as Host B gets the address, it will detect that the address is already in use by someone else (Host A) and will send a DHCPDECLINE message. Therefore, in this situation, the server has to temporarily assign a different address from the dynamic pool (not matching what has been reserved) to Host B.

When Host A renews its address, the server will discover that the address being renewed is now reserved for someone else (Host B). Therefore, the server will remove the lease for 2001:db8::10, select a new address, and create a new lease for it. It will send two addresses in its response: the old address, with lifetime set to 0 to explicitly indicate that it is no longer valid; and the new address, with a non-zero lifetime. When Host B renews its temporarily assigned address, the server will detect that the existing lease does not match the reservation, so it will release the current address Host B has and will create a new lease matching the reservation. As before, the server will send two addresses: the temporarily assigned one with zeroed lifetimes, and the new one that matches the reservation with proper lifetimes set.

This recovery will succeed, even if other hosts attempt to get the reserved address. If Host C requests the address 2001:db8::10 after the reservation is made, the server will propose a different address.

This recovery mechanism allows the server to fully recover from a case where reservations conflict with existing leases. This procedure takes time and will roughly take as long as the value set for renew-timer. The best way to avoid such recovery is not to define new reservations that conflict with existing leases. Another recommendation is to use out-of-pool reservations. If the reserved address does not belong to a pool, there is no way that other clients can get this address.

Note

The conflict-resolution mechanism does not work for global reservations. As of Kea 1.5.0, it is generally recommended not to use global reservations for addresses or prefixes. If you choose to use them anyway, you must manually ensure that the reserved values are not in the dynamic pools.



9.3.3 Reserving a Hostname

When the reservation for a client includes the **hostname**, the server will assign this hostname to the client and send it back in the Client FQDN, if the client sent the FQDN option to the server. The reserved hostname always takes precedence over the hostname supplied by the client (via the FQDN option) or the autogenerated (from the IPv6 address) hostname.

The server qualifies the reserved hostname with the value of the **qualifying-suffix** parameter. For example, the following subnet configuration:

```
"subnet6": [
  {
    "subnet": "2001:db8:1::/48",
    "pools": [ { "pool": "2001:db8:1::/80" } ],
    "reservations": [
      {
        "duid": "01:02:03:04:05:0A:0B:0C:0D:0E",
        "ip-addresses": [ "2001:db8:1::100" ]
        "hostname": "alice-laptop"
      }
    ]
  }
],
"dhcp-ddns": {
  "enable-updates": true,
  "qualifying-suffix": "example.isc.org."
}
```

will result in assigning the "alice-laptop.example.isc.org." hostname to the client using the DUID "01:02:03:04:05:0A:0B:0C:0D:0E". If the **qualifying-suffix** is not specified, the default (empty) value will be used, and in this case the value specified as a **hostname** will be treated as a fully qualified name. Thus, by leaving the **qualifying-suffix** empty it is possible to qualify hostnames for different clients with different domain names:

```
"subnet6": [
  {
    "subnet": "2001:db8:1::/48",
    "pools": [ { "pool": "2001:db8:1::/80" } ],
    "reservations": [
      {
        "duid": "01:02:03:04:05:0A:0B:0C:0D:0E",
        "ip-addresses": [ "2001:db8:1::100" ]
        "hostname": "mark-desktop.example.org."
      }
    ]
  }
],
"dhcp-ddns": {
  "enable-updates": true,
}
```

The above example results in the assignment of the "mark-desktop.example.org." hostname to the client using the DUID "01:02:03:04:05:0A:0B:0C:0D:0E".

9.3.4 Including Specific DHCPv6 Options in Reservations

Kea offers the ability to specify options on a per-host basis. These options follow the same rules as any other options. These can be standard options (see Section 9.2.11), custom options (see Section 9.2.13), or vendor-specific options (see Section 9.2.14). The following example demonstrates how standard options can be defined.

```
"reservations": [
  {
    "duid": "01:02:03:05:06:07:08",
```



```
"ip-addresses": [ "2001:db8:1::2" ],
"option-data": [
  {
    "option-data": [ {
      "name": "dns-servers",
      "data": "3000:1::234"
    },
    {
      "name": "nis-servers",
      "data": "3000:1::234"
    }
  ]
} ]
```

Vendor-specific options can be reserved in a similar manner:

```
"reservations": [
{
  "duid": "aa:bb:cc:dd:ee:ff",
  "ip-addresses": [ "2001:db8::1" ],
  "option-data": [
    {
      "name": "vendor-opts",
      "data": 4491
    },
    {
      "name": "tftp-servers",
      "space": "vendor-4491",
      "data": "3000:1::234"
    }
  ]
} ]
```

Options defined at host level have the highest priority. In other words, if there are options defined with the same type on global, subnet, class, and host level, the host-specific values will be used.

9.3.5 Reserving Client Classes in DHCPv6

Section 14.3 explains how to configure the server to assign classes to a client, based on the content of the options that this client sends to the server. Host reservations mechanisms also allow for the static assignment of classes to clients. The definitions of these classes are placed in the Kea configuration. The following configuration snippet shows how to specify that the client belongs to classes **reserved-class1** and **reserved-class2**. Those classes are associated with specific options being sent to the clients which belong to them.

```
{
  "client-classes": [
    {
      "name": "reserved-class1",
      "option-data": [
        {
          "name": "dns-servers",
          "data": "2001:db8:1::50"
        }
      ]
    },
    {
      "name": "reserved-class2",
      "option-data": [
        {
          "name": "nis-servers",
          "data": "2001:db8:1::100"
        }
      ]
    }
  ]
}
```



```
    }
  ]
},
],
"subnet6": [
  { "pools": [ { "pool": "2001:db8:1::/64" } ],
    "subnet": "2001:db8:1::/48",
    "reservations": [
      {
        "duid": "01:02:03:04:05:06:07:08",

        "client-classes": [ "reserved-class1", "reserved-class2" ]

      } ]
    } ]
} ]
}
```

Static class assignments, as shown above, can be used in conjunction with classification, using expressions. The "KNOWN" or "UNKNOWN" builtin class is added to the packet and any class depending on it (directly or indirectly) and not only-if-required is evaluated.

Note

If you want to force the evaluation of a class expression after the host reservation lookup, for instance because of a dependency on "reserved-class1" from the previous example, you should add a "member('KNOWN')" statement in the expression.

9.3.6 Storing Host Reservations in MySQL, PostgreSQL, or Cassandra

It is possible to store host reservations in MySQL, PostgreSQL, or Cassandra. See Section 9.2.3 for information on how to configure Kea to use reservations stored in MySQL, PostgreSQL, or Cassandra. Kea provides a dedicated hook for managing reservations in a database; section Section 15.4.4 provides detailed information. The Kea wiki <https://gitlab.isc.org/isc-projects/kea/wikis/designs/commands#23-host-reservations-hr-management> provides some examples of how to conduct common host reservations operations.

Note

In Kea, the maximum length of an option specified per-host is arbitrarily set to 4096 bytes.

9.3.7 Fine Tuning DHCPv6 Host Reservation

The host reservation capability introduces additional restrictions for the allocation engine (the component of Kea that selects an address for a client) during lease selection and renewal. In particular, three major checks are necessary. First, when selecting a new lease, it is not sufficient for a candidate lease to simply not be in use by another DHCP client; it also must not be reserved for another client. Second, when renewing a lease, an additional check must be performed to see whether the address being renewed is reserved for another client. Finally, when a host renews an address or a prefix, the server must check whether there is a reservation for this host, so the existing (dynamically allocated) address should be revoked and the reserved one be used instead.

Some of those checks may be unnecessary in certain deployments and not performing them may improve performance. The Kea server provides the **reservation-mode** configuration parameter to select the types of reservations allowed for a particular subnet. Each reservation type has different constraints for the checks to be performed by the server when allocating or renewing a lease for the client. Allowed values are:

- **all** - enables both in-pool and out-of-pool host reservation types. This setting is the default value, and is the safest and most flexible. However, as all checks are conducted, it is also the slowest. It does not check against global reservations.
-



- **out-of-pool** - allows only out of pool host reservations. With this setting in place, the server may assume that all host reservations are for addresses that do not belong to the dynamic pool. Therefore, it can skip the reservation checks when dealing with in-pool addresses, thus improving performance. Do not use this mode if any of your reservations use in-pool addresses. Caution is advised when using this setting; Kea does not sanity-check the reservations against **reservation-mode** and misconfiguration may cause problems.
- **global** - allows only global host reservations. With this setting in place, the server searches for reservations for a client only among the defined global reservations. If an address is specified, the server will skip the reservation checks carried out when dealing in other modes, thus improving performance. Caution is advised when using this setting; Kea does not sanity-check the reservations when **global** and misconfiguration may cause problems.
- **disabled** - host reservation support is disabled. As there are no reservations, the server will skip all checks. Any reservations defined will be completely ignored. As the checks are skipped, the server may operate faster in this mode.

The parameter can be specified at global, subnet, and shared-network levels.

An example configuration that disables reservation looks as follows:

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "reservation-mode": "disabled",
      ...
    }
  ]
}
```

An example configuration using global reservations is shown below:

```
"Dhcp6": {

  "reservation-mode": "global",
  "reservations": [
    {
      "duid": "00:03:00:01:11:22:33:44:55:66",
      "hostname": "host-one"
    },
    {
      "duid": "00:03:00:01:99:88:77:66:55:44",
      "hostname": "host-two"
    }
  ],

  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      ...
    }
  ]
}
```

For more details regarding global reservations, see [Section 9.3.8](#).

Another aspect of the host reservations is the different types of identifiers. Kea currently supports two types of identifiers in DHCPv6: hardware address and DUID. This is beneficial from a usability perspective; however, there is one drawback. For each incoming packet Kea has to extract each identifier type and then query the database to see if there is a reservation by this particular identifier. If nothing is found, the next identifier is extracted and the next query is issued. This process continues until either a reservation is found or all identifier types have been checked. Over time, with an increasing number of supported identifier types, Kea would become slower and slower.



To address this problem, a parameter called **host-reservation-identifiers** has been introduced. It takes a list of identifier types as a parameter. Kea will check only those identifier types enumerated in `host-reservation-identifiers`. From a performance perspective, the number of identifier types should be kept to a minimum, ideally one. If your deployment uses several reservation types, please enumerate them from most- to least-frequently used, as this increases the chances of Kea finding the reservation using the fewest queries. An example of host reservation identifiers looks as follows:

```
"host-reservation-identifiers": [ "duid", "hw-address" ],
"subnet6": [
  {
    "subnet": "2001:db8:1::/64",
    ...
  }
]
```

If not specified, the default value is:

```
"host-reservation-identifiers": [ "hw-address", "duid" ]
```

9.3.8 Global Reservations in DHCPv6

In some deployments, such as mobile, clients can roam within the network and certain parameters must be specified regardless of the client's current location. To facilitate such a need, a global reservation mechanism has been implemented. The idea behind it is that regular host reservations are tied to specific subnets, by using a specific `subnet-id`. Kea can specify a global reservation that can be used in every subnet that has global reservations enabled.

This feature can be used to assign certain parameters, such as hostname or other dedicated, host-specific options. It can also be used to assign addresses or prefixes. However, global reservations that assign either of these bypass the whole topology determination provided by DHCP logic implemented in Kea. It is very easy to misuse this feature and get a configuration that is inconsistent. To give a specific example, imagine a global reservation for an address `2001:db8:1111::1` and two subnets `2001:db8:1111::/48` and `2001:db8:ffff::/48`. If global reservations are used in both subnets and a device matching global host reservations visits part of the network that is covered by `2001:db8:ffff::/48`, it will get an IP address `2001:db8:ffff::1`, which will be outside of the prefix announced by its local router using Router Advertisements. Such a configuration would be unusable or, at the very least, riddled with issues, such as downlink traffic not reaching the device.

To use global host reservations, a configuration similar to the following can be used:

```
"Dhcp6:" {
  // This specifies global reservations. They will apply to all subnets that
  // have global reservations enabled.

  "reservations": [
    {
      "hw-address": "aa:bb:cc:dd:ee:ff",
      "hostname": "hw-host-dynamic"
    },
    {
      "hw-address": "01:02:03:04:05:06",
      "hostname": "hw-host-fixed",

      // Use of IP address in global reservation is risky. If used outside of
      // matching subnet, such as 3001::/64, it will result in a broken
      // configuration being handled to the client.
      "ip-address": "2001:db8:ff::77"
    },
    {
      "duid": "01:02:03:04:05",
      "hostname": "duid-host"
    }
  ],
  "valid-lifetime": 600,
```



```
"subnet4": [ {
  "subnet": "2001:db8:1::/64",
  "reservation-mode": "global",
  "pools": [ { "pool": "2001:db8:1::-2001:db8:1::100" } ]
} ]
}
```

When using database backends, the global host reservations are distinguished from regular reservations by using subnet-id value of zero.

9.4 Shared Networks in DHCPv6

DHCP servers use subnet information in two ways. First, it is used to determine the point of attachment, or simply put, where the client is connected to the network. Second, the subnet information is used to group information pertaining to a specific location in the network. This approach works well in general cases, but there are scenarios where the boundaries are blurred. Sometimes it is useful to have more than one logical IP subnet being deployed on the same physical link. The need to understand that two or more subnets are used on the same link requires additional logic in the DHCP server. This capability is called "shared networks" in Kea and ISC DHCP configurations. It is sometimes also called "shared subnets." In Microsoft's nomenclature it is called "multinet."

There are many use cases where the feature is useful. The most common example in the IPv4 case is when the server is running out of available addresses in a subnet. This is less common in IPv6, but shared networks are still useful in IPv6. One of the use cases is an exhaustion of IPv6- delegated prefixes within a subnet. Another IPv6-specific example is an experiment with an addressing scheme. With the advent of IPv6 deployment and a vast address space, many organizations split the address space into subnets, deploy it, and then after a while discover that they want to split it differently. In the transition period, they want both old and new addressing to be available. Thus the need for more than one subnet on the same physical link.

Finally, the case of cable networks is directly applicable in IPv6. There are two types of devices in cable networks: cable modems and the end-user devices behind them. It is a common practice to use different subnets for cable modems to prevent users from tinkering with them. In this case, the distinction is based on the type of device, rather than address-space exhaustion.

A client connected to a shared network may be assigned a lease (address or prefix) from any of the pools defined within the subnets belonging to the shared network. Internally, the server selects one of the subnets belonging to a shared network and tries to allocate a lease from this subnet. If the server is unable to allocate a lease from the selected subnet (e.g., due to pools exhaustion), it will use another subnet from the same shared network and try to allocate a lease from this subnet, etc. Therefore, in the typical case, the server will allocate all leases available in a given subnet before it starts allocating leases from other subnets belonging to the same shared network. However, in certain situations the client can be allocated a lease from the other subnets before the pools in the first subnet get exhausted, e.g. when the client provides a hint that belongs to another subnet or the client has reservations in a subnet other than the default.

Note

Deployments should not assume that Kea waits until it has allocated all the addresses from the first subnet in a shared network before allocating addresses from other subnets.

In order to define a shared network an additional configuration scope is introduced:

```
{
  "Dhcp6": {
    "shared-networks": [
      {
        // Name of the shared network. It may be an arbitrary string
        // and it must be unique among all shared networks.
        "name": "ipv6-lab-1",

        // The subnet selector can be specified on the shared network level.
        // Subnets from this shared network will be selected for clients
        // communicating via relay agent having the specified IP address.
      }
    ]
  }
}
```



```
"relay": {
  "ip-addresses": [ "2001:db8:2:34::1" ]
},

// This starts a list of subnets in this shared network.
// There are two subnets in this example.
"subnet6": [
  {
    "subnet": "2001:db8::/48",
    "pools": [ { "pool": "2001:db8::1 - 2001:db8::ffff" } ]
  },
  {
    "subnet": "3ffe:ffe::/64",
    "pools": [ { "pool": "3ffe:ffe::/64" } ]
  }
]
} ], // end of shared-networks

// It is likely that in your network you'll have a mix of regular,
// "plain" subnets and shared networks. It is perfectly valid to mix
// them in the same config file.
//
// This is regular subnet. It's not part of any shared-network.
"subnet6": [
  {
    "subnet": "2001:db9::/48",
    "pools": [ { "pool": "2001:db9::/64" } ],
    "relay": {
      "ip-addresses": [ "2001:db8:1:2::1" ]
    }
  }
]
} // end of Dhcp6
}
```

As you see in the example, it is possible to mix shared and regular ("plain") subnets. Each shared network must have a unique name. This is similar to the ID for subnets, but gives administrators more flexibility. This is used for logging, but also internally for identifying shared networks.

In principle it makes sense to define only shared networks that consist of two or more subnets. However, for testing purposes it is allowed to define a shared network with just one subnet or even an empty one. This is not a recommended practice in production networks, as the shared network logic requires additional processing and thus lowers the server's performance. To avoid unnecessary performance degradation, the shared subnets should only be defined when required by the deployment.

Shared networks provide an ability to specify many parameters in the shared network scope that will apply to all subnets within it. If necessary, you can specify a parameter in the shared network scope and then override its value in the subnet scope. For example:

```
"shared-networks": [
  {
    "name": "lab-network3",
    "relay": {
      "ip-addresses": [ "2001:db8:2:34::1" ]
    },

    // This applies to all subnets in this shared network, unless
    // values are overridden on subnet scope.
    "valid-lifetime": 600,

    // This option is made available to all subnets in this shared
    // network.
  }
]
```



```
"option-data": [ {
  "name": "dns-servers",
  "data": "2001:db8::8888"
} ],

"subnet6": [
  {
    "subnet": "2001:db8:1::/48",
    "pools": [ { "pool": "2001:db8:1::1 - 2001:db8:1::ffff" } ],

    // This particular subnet uses different values.
    "valid-lifetime": 1200,
    "option-data": [
      {
        "name": "dns-servers",
        "data": "2001:db8::1:2"
      },
      {
        "name": "unicast",
        "data": "2001:abcd::1"
      }
    ]
  },
  {
    "subnet": "2001:db8:2::/48",
    "pools": [ { "pool": "2001:db8:2::1 - 2001:db8:2::ffff" } ],

    // This subnet does not specify its own valid-lifetime value,
    // so it is inherited from shared network scope.
    "option-data": [
      {
        "name": "dns-servers",
        "data": "2001:db8:cafe::1"
      }
    ]
  }
],
}
```

In this example, there is a `dns-servers` option defined that is available to clients in both subnets in this shared network. Also, a valid lifetime is set to 10 minutes (600s). However, the first subnet overrides some of the values (valid lifetime is 20 minutes, different IP address for `dns-servers`), but also adds its own option (unicast address). Assuming a client asking for a server unicast and dns servers options is assigned a lease from this subnet, it will get a lease for 20 minutes and `dns-servers`, and be allowed to use server unicast at address `2001:abcd::1`. If the same client is assigned to the second subnet, it will get a 10-minute lease, a `dns-servers` value of `2001:db8:cafe::1`, and no server unicast.

Some parameters must be the same in all subnets in the same shared network. This restriction applies to the **interface** and **rapid-commit** settings. The most convenient way is to define them on the shared network scope, but you may specify them for each subnet. However, care should be taken for each subnet to have the same value.

9.4.1 Local and Relayed Traffic in Shared Networks

It is possible to specify an interface name in the shared network scope to tell the server that this specific shared network is reachable directly (not via relays) using a local network interface. It is sufficient to specify it once at the shared network level. As all subnets in a shared network are expected to be used on the same physical link, it is a configuration error to attempt to define a shared network using subnets that are reachable over different interfaces. It is possible to specify the interface parameter on each subnet, although its value must be the same for each subnet. Thus it is usually more convenient to specify it once at the shared network level.

```
"shared-networks": [
  {
    "name": "office-floor-2",
```



```
// This tells Kea that the whole shared networks is reachable over
// local interface. This applies to all subnets in this network.
"interface": "eth0",

"subnet6": [
  {
    "subnet": "2001:db8::/64",
    "pools": [ { "pool": "2001:db8::1 - 2001:db8::ffff" } ],
    "interface": "eth0"
  },
  {
    "subnet": "3ffe:abcd::/64",
    "pools": [ { "pool": "3ffe:abcd::1 - 3ffe:abcd::ffff" } ]

    // Specifying a different interface name is configuration
    // error:
    // "interface": "eth1"
  }
],
} ]
```

Somewhat similar to interface names, relay IP addresses can also be specified for the whole shared network. However, depending on your relay configuration, it may use different IP addresses depending on which subnet is being used. Thus there is no requirement to use the same IP relay address for each subnet. Here's an example:

```
"shared-networks": [
  {
    "name": "kakapo",
    "relay": {
      "ip-addresses": [ "2001:db8::abcd" ]
    },
    "subnet6": [
      {
        "subnet": "2001:db8::/64",
        "relay": {
          "ip-addresses": [ "2001:db8::1234" ]
        },
        "pools": [ { "pool": "2001:db8::1 - 2001:db8::ffff" } ]
      },
      {
        "subnet": "3ffe:abcd::/64",
        "pools": [ { "pool": "3ffe:abcd::1 - 3ffe:abcd::ffff" } ],
        "relay": {
          "ip-addresses": [ "3ffe:abcd::cafe" ]
        }
      }
    ]
  }
]
```

In this particular case the relay IP address specified at the network level doesn't have much sense, as it is overridden in both subnets, but it was left there as an example of how one could be defined at the network level. Note that the relay agent IP address typically belongs to the subnet it relays packets from, but this is not a strict requirement. Kea accepts any value here as long as it is a valid IPv6 address.

9.4.2 Client Classification in Shared Networks

Sometimes it is desirable to segregate clients into specific subnets based on certain properties. This mechanism is called client classification and is described in Chapter 14. Client classification can be applied to subnets belonging to shared networks in the



same way as it is used for subnets specified outside of shared networks. It is important to understand how the server selects subnets for clients when client classification is in use, to ensure that the desired subnet is selected for a given client type.

If a subnet is associated with a class, only the clients belonging to this class can use this subnet. If there are no classes specified for a subnet, any client connected to a given shared network can use this subnet. A common mistake is to assume that the subnet including a client class is preferred over subnets without client classes. Consider the following example:

```
{
  "client-classes": [
    {
      "name": "b-devices",
      "test": "option[1234].hex == 0x0002"
    }
  ],
  "shared-networks": [
    {
      "name": "galah",
      "relay": {
        "ip-address": [ "2001:db8:2:34::1" ]
      },
      "subnet6": [
        {
          "subnet": "2001:db8:1::/64",
          "pools": [ { "pool": "2001:db8:1::20 - 2001:db8:1::ff" } ],
        },
        {
          "subnet": "2001:db8:3::/64",
          "pools": [ { "pool": "2001:db8:3::20 - 2001:db8:3::ff" } ],
          "client-class": "b-devices"
        }
      ]
    }
  ]
}
```

If the client belongs to the "b-devices" class (because it includes option 1234 with a value of 0x0002), that doesn't guarantee that the subnet 2001:db8:3::/64 will be used (or preferred) for this client. The server can use either of the two subnets because the subnet 2001:db8:1::/64 is also allowed for this client. The client classification used in this case should be perceived as a way to restrict access to certain subnets, rather than a way to express subnet preference. For example, if the client doesn't belong to the "b-devices" class it may only use the subnet 2001:db8:1::/64 and will never use the subnet 2001:db8:3::/64.

A typical use case for client classification is in a cable network, where cable modems should use one subnet and other devices should use another subnet within the same shared network. In this case it is necessary to apply classification on all subnets. The following example defines two classes of devices, and the subnet selection is made based on option 1234 values.

```
{
  "client-classes": [
    {
      "name": "a-devices",
      "test": "option[1234].hex == 0x0001"
    },
    {
      "name": "b-devices",
      "test": "option[1234].hex == 0x0002"
    }
  ],
  "shared-networks": [
    {
      "name": "galah",
      "relay": {
        "ip-addresses": [ "2001:db8:2:34::1" ]
      }
    }
  ]
}
```



```
    },
    "subnet6": [
      {
        "subnet": "2001:db8:1::/64",
        "pools": [ { "pool": "2001:db8:1::20 - 2001:db8:1::ff" } ],
        "client-class": "a-devices"
      },
      {
        "subnet": "2001:db8:3::/64",
        "pools": [ { "pool": "2001:db8:3::20 - 2001:db8:3::ff" } ],
        "client-class": "b-devices"
      }
    ]
  }
}
```

In this example each class has its own restriction. Only clients that belong to class "a-devices" will be able to use subnet 2001:db8:1::/64 and only clients belonging to "b-devices" will be able to use subnet 2001:db8:3::/64. Care should be taken not to define too-restrictive classification rules, as clients that are unable to use any subnets will be refused service. However, this may be a desired outcome if one wishes to provide service only to clients with known properties (e.g. only VoIP phones allowed on a given link).

Note that it is possible to achieve an effect similar to the one presented in this section without the use of shared networks. If the subnets are placed in the global subnets scope, rather than in the shared network, the server will still use classification rules to pick the right subnet for a given class of devices. The major benefit of placing subnets within the shared network is that common parameters for the logically grouped subnets can be specified once, in the shared network scope, e.g. the "interface" or "relay" parameter. All subnets belonging to this shared network will inherit those parameters.

9.4.3 Host Reservations in Shared Networks

Subnets that are part of a shared network allow host reservations, similar to regular subnets:

```
{
  "shared-networks": [
    {
      "name": "frog",
      "relay": {
        "ip-addresses": [ "2001:db8:2:34::1" ]
      },
      "subnet6": [
        {
          "subnet": "2001:db8:1::/64",
          "id": 100,
          "pools": [ { "pool": "2001:db8:1::1 - 2001:db8:1::64" } ],
          "reservations": [
            {
              "duid": "00:03:00:01:11:22:33:44:55:66",
              "ip-addresses": [ "2001:db8:1::28" ]
            }
          ]
        },
        {
          "subnet": "2001:db8:3::/64",
          "id": 101,
          "pools": [ { "pool": "2001:db8:3::1 - 2001:db8:3::64" } ],
          "reservations": [
            {
              "duid": "00:03:00:01:aa:bb:cc:dd:ee:ff",
              "ip-addresses": [ "2001:db8:2::28" ]
            }
          ]
        }
      ]
    }
  ]
}
```



```
    }
  ]
}
}
```

It is worth noting that Kea conducts additional checks when processing a packet if shared networks are defined. Instead of simply checking whether there's a reservation for a given client in its initially selected subnet, it looks through all subnets in a shared network for a reservation. This is one of the reasons why defining a shared network may impact performance. If there is a reservation for a client in any subnet, that particular subnet will be picked for the client. Although it's technically not an error, it is considered a bad practice to define reservations for the same host in multiple subnets belonging to the same shared network.

While not strictly mandatory, it is strongly recommended to use explicit "id" values for subnets if you plan to use database storage for host reservations. If ID is not specified, the values for it are autogenerated, i.e. it assigns increasing integer values starting from 1. Thus, the autogenerated IDs are not stable across configuration changes.

9.5 Server Identifier in DHCPv6

The DHCPv6 protocol uses a "server identifier" (also known as a DUID) to allow clients to discriminate between several servers present on the same link. [RFC 8415](#) currently defines four DUID types: DUID-LLT, DUID-EN, DUID-LL, and DUID-UUID.

The Kea DHCPv6 server generates a server identifier once, upon the first startup, and stores it in a file. This identifier is not modified across restarts of the server and so is a stable identifier.

Kea follows the recommendation from [RFC 8415](#) to use DUID-LLT as the default server identifier. However, we have received reports that some deployments require different DUID types, and there is a need to administratively select both DUID type and/or its contents.

The server identifier can be configured using parameters within the **server-id** map element in the global scope of the Kea configuration file. The following example demonstrates how to select DUID-EN as a server identifier:

```
"Dhcp6": {
  "server-id": {
    "type": "EN"
  },
  ...
}
```

Currently supported values for **type** parameter are: "LLT", "EN", and "LL", for DUID-LLT, DUID-EN, and DUID-LL respectively.

When a new DUID type is selected the server generates its value and replaces any existing DUID in the file. The server then uses the new server identifier in all future interactions with the clients.

Note

If the new server identifier is created after some clients have obtained their leases, the clients using the old identifier are not able to renew the leases; the server will ignore messages containing the old server identifier. Clients will continue sending Renew until they transition to the rebinding state. In this state, they will start sending Rebind messages to the multicast address without a server identifier. The server will respond to the Rebind messages with a new server identifier, and the clients will associate the new server identifier with their leases. Although the clients will be able to keep their leases and will eventually learn the new server identifier, this will be at the cost of an increased number of renewals and multicast traffic due to a need to rebind. Therefore, it is recommended that modification of the server identifier type and value is avoided if the server has already assigned leases and these leases are still valid.

There are cases when an administrator needs to explicitly specify a DUID value rather than allow the server to generate it. The following example demonstrates how to explicitly set all components of a DUID-LLT.



```
"Dhcp6": {
  "server-id": {
    "type": "LLT",
    "htype": 8,
    "identifier": "A65DC7410F05",
    "time": 2518920166
  },
  ...
}
```

where:

- **htype** is a 16-bit unsigned value specifying hardware type,
- **identifier** is a link-layer address, specified as a string of hexadecimal digits, and
- **time** is a 32-bit unsigned time value.

The hexadecimal representation of the DUID generated as a result of the configuration specified above will be:

```
00:01:00:08:96:23:AB:E6:A6:5D:C7:41:0F:05
|type|htype|  time  |  identifier  |
```

A special value of 0 for "htype" and "time" is allowed, which indicates that the server should use ANY value for these components. If the server already uses a DUID-LLT it will use the values from this DUID; if the server uses a DUID of a different type or doesn't yet use any DUID, it will generate these values. Similarly, if the "identifier" is assigned an empty string, the value of the identifier will be generated. Omitting any of these parameters is equivalent to setting them to those special values.

For example, the following configuration:

```
"Dhcp6": {
  "server-id": {
    "type": "LLT",
    "htype": 0,
    "identifier": "",
    "time": 2518920166
  },
  ...
}
```

indicates that the server should use ANY link-layer address and hardware type. If the server is already using DUID-LLT, it will use the link-layer address and hardware type from the existing DUID. If the server is not yet using any DUID, it will use the link-layer address and hardware type from one of the available network interfaces. The server will use an explicit value of time; if it is different than a time value present in the currently used DUID, that value will be replaced, effectively modifying the current server identifier.

The following example demonstrates an explicit configuration of a DUID-EN:

```
"Dhcp6": {
  "server-id": {
    "type": "EN",
    "enterprise-id": 2495,
    "identifier": "87ABEF7A5BB545"
  },
  ...
}
```

where:

- **enterprise-id** is a 32-bit unsigned value holding an enterprise number, and



- **identifier** is a variable-length identifier within DUID-EN.

The hexadecimal representation of the DUID-EN created according to the configuration above is:

```
00:02:00:00:09:BF:87:AB:EF:7A:5B:B5:45
|type| ent-id | identifier |
```

As in the case of the DUID-LLT, special values can be used for the configuration of the DUID-EN. If the **enterprise-id** is 0, the server will use a value from the existing DUID-EN. If the server is not using any DUID or the existing DUID has a different type, the ISC enterprise id will be used. When an empty string is entered for **identifier**, the identifier from the existing DUID-EN will be used. If the server is not using any DUID-EN, a new 6-byte-long identifier will be generated.

DUID-LL is configured in the same way as DUID-LLT except that the **time** parameter has no effect for DUID-LL, because this DUID type only comprises a hardware type and link-layer address. The following example demonstrates how to configure DUID-LL:

```
"Dhcp6": {
  "server-id": {
    "type": "LL",
    "htype": 8,
    "identifier": "A65DC7410F05"
  },
  ...
}
```

which will result in the following server identifier:

```
00:03:00:08:A6:5D:C7:41:0F:05
|type| htype| identifier |
```

The server stores the generated server identifier in the following location: [kea-install-dir]/var/lib/kea/kea-dhcp6-serverid.

In some uncommon deployments where no stable storage is available, the server should be configured not to try to store the server identifier. This choice is controlled by the value of the **persist** boolean parameter:

```
"Dhcp6": {
  "server-id": {
    "type": "EN",
    "enterprise-id": 2495,
    "identifier": "87ABEF7A5BB545",
    "persist": false
  },
  ...
}
```

The default value of the "persist" parameter is **true**, which configures the server to store the server identifier on a disk.

In the example above, the server is configured not to store the generated server identifier on a disk. But if the server identifier is not modified in the configuration, the same value will be used after server restart, because the entire server identifier is explicitly specified in the configuration.

9.6 DHCPv6 data directory

The Kea DHCPv6 server puts the server identifier file and the default memory lease file into its data directory. By default this directory is **prefix**/var/lib/kea but this location can be changed using the **data-directory** global parameter as in:

```
"Dhcp6": {
  "data-directory": "/var/tmp/kea-server6",
  ...
}
```



9.7 Stateless DHCPv6 (Information-Request Message)

Typically DHCPv6 is used to assign both addresses and options. These assignments (leases) have a state that changes over time, hence their name, stateful. DHCPv6 also supports a stateless mode, where clients request configuration options only. This mode is considered lightweight from the server perspective, as it does not require any state tracking, and carries the stateless name.

The Kea server supports stateless mode. Clients can send Information-Request messages and the server sends back answers with the requested options (providing the options are available in the server configuration). The server attempts to use per-subnet options first. If that fails - for whatever reason - it then tries to provide options defined in the global scope.

Stateless and stateful mode can be used together. No special configuration directives are required to handle this; simply use the configuration for stateful clients and the stateless clients will get just the options they requested.

This usage of global options allows for an interesting case. It is possible to run a server that provides only options and no addresses or prefixes. If the options have the same value in each subnet, the configuration can define required options in the global scope and skip subnet definitions altogether. Here's a simple example of such a configuration:

```
"Dhcp6": {
  "interfaces-config": {
    "interfaces": [ "ethX" ]
  },
  "option-data": [ {
    "name": "dns-servers",
    "data": "2001:db8::1, 2001:db8::2"
  } ],
  "lease-database": {
    "type": "memfile"
  }
}
```

This very simple configuration will provide DNS server information to all clients in the network, regardless of their location. Note the specification of the memfile lease database; this is needed as Kea requires a lease database to be specified even if it is not used.

9.8 Support for RFC 7550 (now part of RFC 8415)

[RFC 7550](#) introduced some changes to the previous DHCPv6 specifications, [RFC 3315](#) and [RFC 3633](#), to resolve a few issues with the coexistence of multiple stateful options in the messages sent between the clients and servers. Those changes were later included in the most recent DHCPv6 protocol specification, [RFC 8415](#), which obsoleted [RFC 7550](#). Kea supports [RFC 8415](#) along with these protocol changes, which are briefly described below.

When a client, such as a requesting router, requests an allocation of both addresses and prefixes during the 4-way (SARR) exchange with the server, and the server is not configured to allocate any prefixes but it can allocate some addresses, it will respond with the IA_NA(s) containing allocated addresses and the IA_PD(s) containing the NoPrefixAvail status code. According to the updated specifications, if the client can operate without prefixes it should accept allocated addresses and transition to the "bound" state. When the client subsequently sends Renew/Rebind messages to the server, according to the T1 and T2 times, to extend the lifetimes of the allocated addresses, and if the client is still interested in obtaining prefixes from the server, it may also include an IA_PD in the Renew/Rebind to request allocation of the prefixes. If the server still cannot allocate the prefixes, it will respond with the IA_PD(s) containing the NoPrefixAvail status code. However, if the server can allocate the prefixes it will allocate and send them in the IA_PD(s) to the client. A similar situation occurs when the server is unable to allocate addresses for the client but can delegate prefixes. The client may request allocation of the addresses while renewing the delegated prefixes. Allocating leases for other IA types while renewing existing leases is by default supported by the Kea DHCPv6 server, and the server provides no configuration mechanisms to disable this behavior.

The following are the other behaviors first introduced in [RFC 7550](#) (now part of [RFC 8415](#)) and supported by the Kea DHCPv6 server:

- Set T1/T2 timers to the same value for all stateful (IA_NA and IA_PD) options to facilitate renewal of all a client's leases at the same time (in a single message exchange),



- Place NoAddrAvail and NoPrefixAvail status codes in the IA_NA and IA_PD options in the Advertise message, rather than as the top-level options.

9.9 Using a Specific Relay Agent for a Subnet

The relay must have an interface connected to the link on which the clients are being configured. Typically the relay has a global IPv6 address configured on the interface that belongs to the subnet from which the server will assign addresses. Normally, the server is able to use the IPv6 address inserted by the relay (in the link-addr field in RELAY-FORW message) to select the appropriate subnet.

However, that is not always the case. The relay address may not match the subnet in certain deployments. This usually means that there is more than one subnet allocated for a given link. The two most common examples where this is the case are long-lasting network renumbering (where both old and new address space is still being used) and a cable network. In a cable network, both cable modems and the devices behind them are physically connected to the same link, yet they use distinct addressing. In such a case, the DHCPv6 server needs additional information (like the value of the interface-id option or the IPv6 address inserted in the link-addr field in the RELAY-FORW message) to properly select an appropriate subnet.

The following example assumes that there is a subnet 2001:db8:1::/64 that is accessible via a relay that uses 3000::1 as its IPv6 address. The server is able to select this subnet for any incoming packets that come from a relay with an address in 2001:db8:1::/64 subnet. It also selects that subnet for a relay with address 3000::1.

```
"Dhcp6": {
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "pools": [
        {
          "pool": "2001:db8:1::1-2001:db8:1::ffff"
        }
      ],
      "relay": {
        "ip-addresses": [ "3000::1" ]
      }
    }
  ]
}
```

If "relay" is specified, the "ip-addresses" parameter within it is mandatory.

Note

The current version of Kea uses the "ip-addresses" parameter, which supports specifying a list of addresses.

9.10 Segregating IPv6 Clients in a Cable Network

In certain cases, it is useful to mix relay address information, introduced in Section 9.9, with client classification, explained in Chapter 14. One specific example is in a cable network, where typically modems get addresses from a different subnet than all devices connected behind them.

Let us assume that there is one CMTS (Cable Modem Termination System) with one CM MAC (a physical link that modems are connected to). We want the modems to get addresses from the 3000::/64 subnet, while everything connected behind the modems should get addresses from another subnet (2001:db8:1::/64). The CMTS that acts as a relay uses address 3000::1. The following configuration can serve that configuration:

```
"Dhcp6": {
  "subnet6": [
    {
```



```
    "subnet": "3000::/64",
    "pools": [
      { "pool": "3000::2 - 3000::ffff" }
    ],
    "client-class": "VENDOR_CLASS_docsis3.0",
    "relay": {
      "ip-addresses": [ "3000::1" ]
    }
  },
  {
    "subnet": "2001:db8:1::/64",
    "pools": [
      {
        "pool": "2001:db8:1::1-2001:db8:1::ffff"
      }
    ],
    "relay": {
      "ip-addresses": [ "3000::1" ]
    }
  }
]
}
```

9.11 MAC/Hardware Addresses in DHCPv6

MAC/hardware addresses are available in DHCPv4 messages from the clients, and administrators frequently use that information to perform certain tasks like per-host configuration and address reservation for specific MAC addresses. Unfortunately, the DHCPv6 protocol does not provide any completely reliable way to retrieve that information. To mitigate that issue, a number of mechanisms have been implemented in Kea. Each of these mechanisms works in certain cases, but may fail in others. Whether the mechanism works in a particular deployment is somewhat dependent on the network topology and the technologies used.

Kea allows specification of which of the supported methods should be used and in what order. This configuration may be considered a fine tuning of the DHCP deployment. In a typical deployment the default value of **"any"** is sufficient and there is no need to select specific methods. Changing the value of this parameter is the most useful in cases when an administrator wants to disable certain methods; for example, if the administrator trusts the network infrastructure more than the information provided by the clients themselves, they may prefer information provided by the relays over that provided by the clients.

The configuration is controlled by the **mac-sources** parameter as follows:

```
"Dhcp6": {
  "mac-sources": [ "method1", "method2", "method3", ... ],

  "subnet6": [ ... ],

  ...
}
```

When not specified, a special value of **"any"** is used, which instructs the server to attempt to try all the methods in sequence and use the value returned by the first one that succeeds. If specified, it must have at least one value.

Supported methods are:

- **any** - Not an actual method, just a keyword that instructs Kea to try all other methods and use the first one that succeeds. This is the default operation if no **mac-sources** are defined.
- **raw** - In principle, a DHCPv6 server could use raw sockets to receive incoming traffic and extract MAC/hardware address information. This is currently not implemented for DHCPv6 and this value has no effect.



- **duid** - DHCPv6 uses DUID identifiers instead of MAC addresses. There are currently four DUID types defined, and two of them (DUID-LLT, which is the default, and DUID-LL) convey MAC address information. Although RFC 8415 forbids it, it is possible to parse those DUIDs and extract necessary information from them. This method is not completely reliable, as clients may use other DUID types, namely DUID-EN or DUID-UUID.
- **ipv6-link-local** - Another possible acquisition method comes from the source IPv6 address. In typical usage, clients are sending their packets from IPv6 link-local addresses. There is a good chance that those addresses are based on EUI-64, which contains a MAC address. This method is not completely reliable, as clients may use other link-local address types. In particular, privacy extensions, defined in RFC 4941, do not use MAC addresses. Also note that successful extraction requires that the address's u-bit must be set to 1 and its g-bit set to 0, indicating that it is an interface identifier as per RFC 2373, section 2.5.1.
- **client-link-addr-option** - One extension defined to alleviate missing MAC issues is the client link-layer address option, defined in RFC 6939. This is an option that is inserted by a relay and contains information about a client's MAC address. This method requires a relay agent that supports the option and is configured to insert it. This method is useless for directly connected clients. This parameter can also be specified as **rfc6939**, which is an alias for **client-link-addr-option**.
- **remote-id** - RFC 4649 defines a remote-id option that is inserted by a relay agent. Depending on the relay agent configuration, the inserted option may convey the client's MAC address information. This parameter can also be specified as **rfc4649**, which is an alias for **remote-id**.
- **subscriber-id** - Another option that is somewhat similar to the previous one is subscriber-id, defined in RFC 4580. It, too, is inserted by a relay agent that is configured to insert it. This parameter can also be specified as **rfc4580**, which is an alias for **subscriber-id**. This method is currently not implemented.
- **docsis-cmts** - Yet another possible source of MAC address information are the DOCSIS options inserted by a CMTS that acts as a DHCPv6 relay agent in cable networks. This method attempts to extract MAC address information from suboption 1026 (cm mac) of the vendor-specific option with vendor-id=4491. This vendor option is extracted from the relay-forward message, not the original client's message.
- **docsis-modem** - The final possible source of MAC address information are the DOCSIS options inserted by the cable modem itself. This method attempts to extract MAC address information from suboption 36 (device id) of the vendor-specific option with vendor-id=4491. This vendor option is extracted from the original client's message, not from any relay options.

Empty `mac-sources` is not allowed. If you do not want to specify it, either simply omit the `mac-sources` definition or specify it with the "any" value which is the default.

9.12 Duplicate Addresses (DECLINE Support)

The DHCPv6 server is configured with a certain pool of addresses that it is expected to hand out to DHCPv6 clients. It is assumed that the server is authoritative and has complete jurisdiction over those addresses. However, for various reasons, such as misconfiguration or a faulty client implementation that retains its address beyond the valid lifetime, there may be devices connected that use those addresses without the server's approval or knowledge.

Such an unwelcome event can be detected by legitimate clients (using Duplicate Address Detection) and reported to the DHCPv6 server using a DHCPDECLINE message. The server will do a sanity check (to see whether the client declining an address really was supposed to use it), and then will conduct a clean-up operation and confirm it by sending back a REPLY message. Any DNS entries related to that address will be removed, the fact will be logged, and hooks will be triggered. After that is complete, the address will be marked as declined (which indicates that it is used by an unknown entity and thus not available for assignment) and a probation time will be set on it. Unless otherwise configured, the probation period lasts 24 hours; after that period, the server will recover the lease (i.e. put it back into the available state) and the address will be available for assignment again. It should be noted that if the underlying issue of a misconfigured device is not resolved, the duplicate-address scenario will repeat. If reconfigured correctly, this mechanism provides an opportunity to recover from such an event automatically, without any system administrator intervention.

To configure the decline probation period to a value other than the default, the following syntax can be used:

```
"Dhcp6": {  
  "decline-probation-period": 3600,  
  "subnet6": [ ... ],
```



```
    ...  
}
```

The parameter is expressed in seconds, so the example above will instruct the server to recycle declined leases after one hour.

There are several statistics and hook points associated with the Decline handling procedure. The `lease6_decline` hook is triggered after the incoming DHCPDECLINE message has been sanitized and the server is about to decline the lease. The `declined-addresses` statistic is increased after the hook returns (both global and subnet-specific variants). (See Section 9.13 and Chapter 15 for more details on DHCPv6 statistics and Kea hook points.)

Once the probation time elapses, the declined lease is recovered using the standard expired-lease reclamation procedure, with several additional steps. In particular, both `declined-addresses` statistics (global and subnet-specific) are decreased. At the same time, `reclaimed-declined-addresses` statistics (again in two variants, global and subnet-specific) are increased.

A note about statistics: The server does not decrease the `assigned-addresses` statistics when a DHCPDECLINE message is received and processed successfully. While technically a declined address is no longer assigned, the primary usage of the `assigned-addresses` statistic is to monitor pool utilization. Most people would forget to include `declined-addresses` in the calculation, and simply use `assigned-addresses/total-addresses`. This would cause a bias towards under-representing pool utilization. As this has a potential for major issues, we decided not to decrease `assigned-addresses` immediately after receiving DHCPDECLINE, but to do it later when Kea recovers the address back to the available pool.

9.13 Statistics in the DHCPv6 Server

Note

This section describes DHCPv6-specific statistics. For a general overview and usage of statistics, see Chapter 16.

The DHCPv6 server supports the following statistics:

9.14 Management API for the DHCPv6 Server

The management API allows the issuing of specific management commands, such as statistics retrieval, reconfiguration, or shutdown. For more details, see Chapter 17. Currently, the only supported communication channel type is UNIX stream socket. By default there are no sockets open; to instruct Kea to open a socket, the following entry in the configuration file can be used:

```
"Dhcp6": {  
  "control-socket": {  
    "socket-type": "unix",  
    "socket-name": "/path/to/the/unix/socket"  
  },  
  
  "subnet6": [  
    ...  
  ],  
  ...  
}
```

The length of the path specified by the `socket-name` parameter is restricted by the maximum length for the UNIX socket name on your operating system, i.e. the size of the `sun_path` field in the `sockaddr_un` structure, decreased by 1. This value varies on different operating systems between 91 and 107 characters. Typical values are 107 on Linux and 103 on FreeBSD.

Communication over the control channel is conducted using JSON structures. See the `Control Channel` section in the *Kea Developer's Guide* for more details.

The DHCPv6 server supports the following operational commands:

- `build-report`
-



Statistic	Data Type	Description
pkt6-received	integer	Number of DHCPv6 packets received. This includes all packets: valid, bogus, corrupted, rejected, etc. This statistic is expected to grow rapidly.
pkt6-receive-drop	integer	Number of incoming packets that were dropped. The exact reason for dropping packets is logged, but the most common reasons may be: an unacceptable or not supported packet type is received, direct responses are forbidden, the server-id sent by the client does not match the server's server-id, or the packet is malformed.
pkt6-parse-failed	integer	Number of incoming packets that could not be parsed. A non-zero value of this statistic indicates that the server received a malformed or truncated packet. This may indicate problems in your network, faulty clients, faulty relay agents, or a bug in the server.
pkt6-solicit-received	integer	Number of SOLICIT packets received. This statistic is expected to grow; its increase means that clients that just booted started their configuration process and their initial packets reached your Kea server.
pkt6-advertise-received	integer	Number of ADVERTISE packets received. Advertise packets are sent by the server and the server is never expected to receive them. A non-zero value of this statistic indicates an error occurring in the network. One likely cause would be a misbehaving relay agent that incorrectly forwards ADVERTISE messages towards the server, rather than back to the clients.
pkt6-request-received	integer	Number of DHCPREQUEST packets received. This statistic is expected to grow. Its increase means that clients that just booted received the server's response (DHCPADVERTISE) and accepted it, and are now requesting an address (DHCPREQUEST).
pkt6-reply-received	integer	Number of REPLY packets received. This statistic is expected to remain zero at all times, as REPLY packets are sent by the server and the server is never expected to receive them. A non-zero value indicates an error. One likely cause would be a misbehaving relay agent that incorrectly forwards REPLY messages towards the server, rather than back to the clients.
pkt6-renew-received	integer	Number of RENEW packets received. This statistic is expected to grow; its increase means that clients received their addresses and prefixes and are trying to renew them.
		Number of REBIND packets received.
pkt6-rebind-received	integer	A non-zero value indicates that clients didn't receive responses to their RENEW messages (through the regular lease-renewal mechanism) and



- config-get
- config-reload
- config-set
- config-test
- config-write
- dhcp-disable
- dhcp-enable
- leases-reclaim
- list-commands
- shutdown
- version-get

as described in Section [17.3](#). In addition, it supports the following statistics-related commands:

- statistic-get
- statistic-reset
- statistic-remove
- statistic-get-all
- statistic-reset-all
- statistic-remove-all

as described in Section [16.3](#).

9.15 User Contexts in IPv6

Kea allows loading hook libraries that sometimes could benefit from additional parameters. If such a parameter is specific to the whole library, it is typically defined as a parameter for the hook library. However, sometimes there is a need to specify parameters that are different for each pool.

User contexts can store arbitrary data as long as it has valid JSON syntax and its top level element is a map (i.e. the data must be enclosed in curly brackets). However, some hook libraries may expect specific formatting; please consult the specific hook library documentation for details.

User contexts can be specified at global scope, shared network, subnet, pool, client class, option data, or definition level, and via host reservation. One other useful usage is the ability to store comments or descriptions.

Let's consider a lightweight 4over6 deployment as an example. It is an IPv6 transition technology that allows mapping IPv6 prefixes into full or partial IPv4 addresses. In the DHCP context, these are specific parameters that are supposed to be delivered to clients in the form of additional options. Values of these options are correlated to delegated prefixes, so it is reasonable to keep these parameters together with the PD pool. On the other hand, lightweight 4over6 is not a commonly used feature, so it is not a part of the base Kea code. The solution to this problem is to use user context. For each PD pool that is expected to be used for lightweight 4over6, a user context with extra parameters is defined. Those extra parameters will be used by a hook library that would be loaded only when dynamic calculation of the lightweight 4over6 option is actually needed. An example configuration looks as follows:



```
"Dhcp6": {
  "subnet6": [ {
    "pd-pools": [
      {
        "prefix": "2001:db8::",
        "prefix-len": 56,
        "delegated-len": 64,

        // This is a pool specific context.
        "user-context": {
          "threshold-percent": 85,
          "v4-network": "192.168.0.0/16",
          "v4-overflow": "10.0.0.0/16",
          "lw4over6-sharing-ratio": 64,
          "lw4over6-v4-pool": "192.0.2.0/24",
          "lw4over6-sysports-exclude": true,
          "lw4over6-bind-prefix-len": 56
        }
      }
    ],
    "subnet": "2001:db8::/32",

    // This is a subnet specific context. You can put any type of
    // information here as long as it is a valid JSON.
    "user-context": {
      "comment": "Those v4-v6 migration technologies are tricky.",
      "experimental": true,
      "billing-department": 42,
      "contact-points": [ "Alice", "Bob" ]
    }
  } ],
  ...
}
```

Kea does not interpret or use the content of the user context; it simply stores it, making it available to the hook libraries. It is up to each hook library to extract the information and use it. The parser translates a "comment" entry into a user context with the entry, which allows a comment to be attached inside the configuration itself.

For more background information, see Section [15.5](#).

9.16 Supported DHCPv6 Standards

The following standards are currently supported:

- *Dynamic Host Configuration Protocol for IPv6*, [RFC 3315](#): Supported messages are SOLICIT, ADVERTISE, REQUEST, RELEASE, RENEW, REBIND, INFORMATION-REQUEST, CONFIRM and REPLY.
- *IPv6 Prefix Options for Dynamic Host Configuration Protocol (DHCP) version 6*, [RFC 3633](#): Supported options are IA_PD and IA_PREFIX. Also supported is the status code NoPrefixAvail.
- *DNS Configuration options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*, [RFC 3646](#): Supported option is DNS_SERVERS.
- *The Dynamic Host Configuration Protocol for IPv6 (DHCPv6) Relay Agent Remote-ID Option*, [RFC 4649](#): REMOTE-ID option is supported.
- *The Dynamic Host Configuration Protocol for IPv6 (DHCPv6) Client Fully Qualified Domain Name (FQDN) Option*, [RFC 4704](#): Supported option is CLIENT_FQDN.
- *Dynamic Host Configuration Protocol for IPv6 (DHCPv6) Option for Dual-Stack Lite*, [RFC 6334](#): the AFTR-Name DHCPv6 Option is supported.



- *Relay-Supplied DHCP Options*, [RFC 6422](#): Full functionality is supported: `OPTION_RSOO`, ability of the server to echo back the options, checks whether an option is RSOO-enabled, ability to mark additional options as RSOO-enabled.
- *Prefix Exclude Option for DHCPv6-based Prefix Delegation*, [RFC 6603](#): Prefix Exclude option is supported.
- *Client Link-Layer Address Option in DHCPv6*, [RFC 6939](#): Supported option is client link-layer address option.
- *Issues and Recommendations with Multiple Stateful DHCPv6 Options*, [RFC 7550](#): All recommendations related to the DHCPv6 server operation are supported.
- *DHCPv6 Options for Configuration of Software Address and Port-Mapped Clients*, [RFC 7598](#): All options indicated in this specification are supported by the DHCPv6 server.
- *Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*, [RFC 8415](#): New DHCPv6 protocol specification which obsoletes RFC 3315, RFC 3633, RFC 3736, RFC 4242, RFC 7083, RFC 7283, and RFC 7550

9.17 DHCPv6 Server Limitations

These are the current limitations of the DHCPv6 server software. Most of them are reflections of the current stage of development and should be treated as “not implemented yet”, rather than actual limitations.

- The server will allocate, renew, or rebind a maximum of one lease for a particular IA option (`IA_NA` or `IA_PD`) sent by a client. [RFC 8415](#) allows for multiple addresses or prefixes to be allocated for a single IA.
- Temporary addresses are not supported.
- Client reconfiguration (`RECONFIGURE`) is not yet supported.

9.18 Kea DHCPv6 server examples

A collection of simple-to-use examples for the DHCPv6 component of Kea is available with the source files, located in the `doc/examples/kea6` directory.

9.19 Configuration Backend in DHCPv6

In the Section [5.2](#) we have described the Configuration Backend feature, its applicability and limitations. This section focuses on the usage of the CB with the DHCPv6 server. It lists the supported parameters, describes limitations and gives examples of the DHCPv6 server configuration to take advantage of the CB. Please also refer to the sibling section [Section 8.14](#) for the DHCPv4 specific usage of the CB.

9.19.1 Supported Parameters

The ultimate goal for the CB is to serve as a central configuration repository for one or multiple Kea servers connected to the database. In the future it will be possible to store the most of the server’s configuration in the database and reduce the configuration file to bare minimum, i.e. the only mandatory parameter will be the **config-control** which includes the necessary information to connect to the database. In the Kea 1.6.0 release, however, only the subset of the DHCPv4 server parameters can be stored in the database. All other parameters must be specified in the JSON configuration file, if required.

The following table lists DHCPv6 specific parameters supported by the Configuration Backend with an indication on which level of the hierarchy it is currently supported. The "n/a" is used in cases when the particular parameter is not applicable on the particular level of the hierarchy or in cases when the parameter is not supported by the server on this level of hierarchy. The "no" is used when the parameter is supported by the server on the particular level of hierarchy but is not configurable via the Configuration Backend.



All supported parameters can be configured via `cb_cmds` hooks library described in the Section 15.4.8. The general rule is that the scalar global parameters are set using the `remote-global-parameter6-set`. The shared network specific parameters are set using the `remote-network6-set`. Finally, the subnet and pool level parameters are set using the `remote-subnet6-set`. Whenever there is an exception from this general rule, it is highlighted in the table. The non-scalar global parameters have dedicated commands, e.g. modifying the global DHCPv6 options (`option-data`) is performed using the `remote-option6-global-set`.

Parameter	Global	Shared Network	Subnet	Pool	Prefix Delegation Pool
calculate-tee-times	yes	yes	yes	n/a	n/a
client-class	n/a	yes	yes	no	no
decline-probation-period	yes	n/a	n/a	n/a	n/a
delegated-len	n/a	n/a	n/a	n/a	yes
dhcp4o6-port	yes	n/a	n/a	n/a	n/a
excluded-prefix	n/a	n/a	n/a	n/a	no
excluded-prefix-len	n/a	n/a	n/a	n/a	no
interface	n/a	yes	yes	n/a	n/a
interface-id	n/a	yes	yes	n/a	n/a
option-data	yes (via remote-option6-global-set)	yes	yes	yes	yes
option-def	yes (via remote-option-def6-set)	n/a	n/a	n/a	n/a
preferred-lifetime	yes	yes	yes	n/a	n/a
prefix	n/a	n/a	n/a	n/a	yes
prefix-len	n/a	n/a	n/a	n/a	yes
rapid-commit	yes	yes	yes	n/a	n/a
rebind-timer	yes	yes	yes	n/a	n/a
relay	n/a	yes	yes	n/a	n/a
renew-timer	yes	yes	yes	n/a	n/a
require-client-classes	n/a	yes	yes	no	no
reservation-mode	yes	yes	yes	n/a	n/a
t1-percent	yes	yes	yes	n/a	n/a
t2-percent	yes	yes	yes	n/a	n/a
valid-lifetime	yes	yes	yes	n/a	n/a

Table 9.4: List of DHCPv6 Parameters Supported by the Configuration Backend

9.19.2 Enabling Configuration Backend

The following configuration snippet demonstrates how to enable the MySQL Configuration Backend for the DHCPv6 server:

```
{
  "Dhcp6": {
    "config-control": {
      "config-databases": [
        {
          "type": "mysql",
          "name": "kea",
          "user": "kea",
          "password": "kea",
          "host": "2001:db8:1::1",
          "port": 3302
        }
      ]
    }
  }
}
```



```
    }
  ],
  "config-fetch-wait-time": 20
},
"hooks-libraries": [
  {
    "library": "/usr/local/lib/kea/hooks/libdhcp_mysql_cb.so"
  },
  {
    "library": "/usr/local/lib/kea/hooks/libdhcp_cb_cmds.so"
  }
],
...
}
}
```

The configuration structure is almost identical as for the DHCPv4 server (see Section [8.14.2](#) for the detailed description).



Chapter 10

Lease Expiration in DHCPv4 and DHCPv6

The primary role of the DHCP server is to assign addresses and/or delegate prefixes to DHCP clients. These addresses and prefixes are often referred to as "leases." Leases are typically assigned to clients for a finite amount of time, known as the "valid lifetime." DHCP clients who wish to continue using their assigned leases will periodically renew them by sending the appropriate message to the DHCP server. The DHCP server records the time when these leases are renewed and calculates new expiration times for them.

If the client does not renew a lease before its valid lifetime elapses, the lease is considered expired. There are many situations when the client may cease lease renewals; a common scenario is when the machine running the client shuts down for an extended period of time.

The process through which the DHCP server makes expired leases available for reassignment is referred to as "lease reclamation" and expired leases returned to availability through this process are referred to as "reclaimed." The DHCP server attempts to reclaim an expired lease as soon as it detects that it has expired. The server has several possible ways to detect expiration: it may attempt to allocate a lease to a client but find this lease already present in the database and expired; or it can periodically query the lease database for expired leases. Regardless of how an expired lease is detected, it must be reclaimed before it can be assigned to a client.

This chapter explains how to configure the server to periodically query for the expired leases, and how to minimize the impact of the periodic lease reclamation process on the server's responsiveness. Finally, it explains "lease affinity," which provides the means to assign the same lease to a returning client after its lease has expired.

Although all configuration examples in this section are provided for the DHCPv4 server, the same parameters may be used for DHCPv6 server configuration.

10.1 Lease Reclamation

Lease reclamation is the process through which an expired lease becomes available for assignment to the same or a different client. This process involves the following steps for each reclaimed lease:

- Invoke callouts for the **lease4_expire** or **lease6_expire** hook points, if hook libraries supporting those callouts are currently loaded.
- Update the DNS, i.e. remove any DNS entries associated with the expired lease.
- Update lease information in the lease database to indicate that the lease is now available for re-assignment.
- Update counters on the server, a process that includes increasing the number of reclaimed leases and decreasing the number of assigned addresses or delegated prefixes.

Please refer to Chapter 12 to see how to configure DNS updates in Kea, and to Chapter 15 for information about using hooks libraries.



10.2 Lease Reclamation Configuration Parameters

The following list presents all configuration parameters pertaining to processing expired leases with their default values:

- **reclaim-timer-wait-time** - This parameter governs intervals between completion of previous reclamation cycle and a start of the next one. Specified in seconds. The default value is 10 [seconds].
- **flush-reclaimed-timer-wait-time** - This parameter controls how often the server initiates lease reclamation procedure. Expressed in seconds. The default value is 25 [seconds].
- **hold-reclaimed-time** - This parameter governs how long the lease should be kept after it was reclaimed. This enables lease affinity when set to a non-zero value. Expressed in seconds. The default value is 3600 [seconds].
- **max-reclaim-leases** - This parameter specifies the maximum number of reclaimed leases that can be processed in one go. Zero means unlimited (i.e. process all reclaimed leases). The default value is 100.
- **max-reclaim-time** - This parameter specifies an upper bound of how long a lease reclamation procedure can take. Zero means no time limit. Expressed in milliseconds. The default value is 250 [milliseconds].
- **unwarned-reclaim-cycles** - If lease reclamation limits are specified (**max-reclaim-leases** and/or **max-reclaim-time**), then under certain circumstances the server may not be able to deal with leases to be reclaimed fast enough. This parameter specifies how many consecutive clean up cycles must end with remaining leases to be processed before a warning is printed. The default is 5 [cycles].

The parameters are explained in more detail in the rest of this chapter.

The default value for any parameter is used when this parameter is not explicitly specified in the configuration. Also, the **expired-leases-processing** map may be omitted entirely in the configuration, in which case the default values are used for all parameters listed above.

10.3 Configuring Lease Reclamation

Kea can be configured to periodically detect and reclaim expired leases. During this process the lease entries in the database are modified or removed. While this is happening, the server will not process incoming DHCP messages to avoid issues with concurrent access to database information. As a result, the server will be unresponsive while lease reclamation is performed and DHCP queries will accumulate; responses will be sent once the lease-reclamation cycle is complete.

In deployments where response time is critical, administrators may wish to minimize the interruptions in service caused by lease reclamation. Toward this end, Kea provides configuration parameters to control the frequency of lease reclamation cycles, the maximum number of leases processed in a single reclamation cycle, and the maximum amount of time a single reclamation cycle is allowed to run before being interrupted. The following examples demonstrate how these parameters can be used:

```
"Dhcp4": {  
  ...  
  "expired-leases-processing": {  
    "reclaim-timer-wait-time": 5,  
    "max-reclaim-leases": 0,  
    "max-reclaim-time": 0,  
  },  
  ...  
}
```

The first parameter is expressed in seconds and specifies an interval between the two consecutive lease reclamation cycles. This is explained by the following diagram:



This diagram shows four lease-reclamation cycles (c1 through c4) of variable duration. Note that the duration of the reclamation cycle depends on the number of expired leases detected and processed in the particular cycle. This duration is usually significantly shorter than the interval between the cycles.

According to the **reclaim-timer-wait-time**, the server keeps fixed intervals of five seconds between the end of one cycle and the start of the next cycle. This guarantees the presence of 5s-long periods during which the server remains responsive to DHCP queries and does not perform lease reclamation. The **max-reclaim-leases** and **max-reclaim-time** are set to 0, which sets no restriction on the maximum number of leases reclaimed in the particular cycle, or on the maximum duration of each cycle.

In deployments with high lease-pool utilization, relatively short valid lifetimes, and frequently disconnecting clients which allow leases to expire, the number of expired leases requiring reclamation at any given time may rise significantly. In this case, it is often desirable to apply restrictions to the maximum duration of a reclamation cycle or the maximum number of leases reclaimed in a cycle. The following configuration demonstrates how this can be done:

```
"Dhcp4": {
  ...

  "expired-leases-processing": {
    "reclaim-timer-wait-time": 3,
    "max-reclaim-leases": 100,
    "max-reclaim-time": 50,
    "unwarned-reclaim-cycles": 10,
  },
  ...
}
```

The **max-reclaim-leases** parameter limits the number of leases reclaimed in a single cycle to 100. The **max-reclaim-time** limits the maximum duration of each cycle to 50ms. The lease-reclamation cycle will be interrupted if either of these limitations is reached. The reclamation of any unreclaimed leases will be attempted in subsequent cycles.

The following diagram illustrates the behavior of the system in the presence of many expired leases, when the limits are applied for the reclamation cycles:



This diagram demonstrates the case when each reclamation cycle takes more than 50ms, and thus is interrupted according to the value of the **max-reclaim-time**. This results in equal durations of all reclamation cycles over time. Note that in this example the limitation of the maximum 100 leases is not reached. This may be the case when database transactions are slow or callouts in the hook libraries attached to the server are slow. Regardless, the chosen values for either the maximum number of leases or a maximum cycle time strongly depend on the particular deployment, the lease database backend being used, and any hooks libraries, etc. Administrators may need to experiment to tune the system to suit the dynamics of their deployment.

It is important to realize that with the use of these limits, there is a risk that expired leases will accumulate faster than the server can reclaim them. This should not be a problem if the server is dealing with a temporary burst of expirations, because it should be able to eventually deal with them over time. However, if leases expire at a high rate for a longer period of time, the unreclaimed leases will pile up in the database. To notify the administrator that the current configuration does not satisfy the needs for reclamation of expired leases, the server issues a warning message in the log if it is unable to reclaim all leases within several



reclamation cycles. The number of cycles after which such a warning is issued is specified with the **unwarned-reclaim-cycles** configuration parameter.

Setting the **reclaim-timer-wait-time** to 0 disables periodic reclamation of the expired leases.

10.4 Configuring Lease Affinity

Suppose that a laptop goes into sleep mode after a period of user inactivity. While the laptop is in sleep mode, its DHCP client will not renew leases obtained from the server and these leases will eventually expire. When the laptop wakes up, it is often desirable for it to continue using its previous assigned IP addresses. To facilitate this, the server needs to correlate returning clients with their expired leases. When the client returns, the server will first check for those leases and re-assign them if they have not been assigned to another client. The ability of the server to re-assign the same lease to a returning client is referred to as "lease affinity."

When lease affinity is enabled (i.e. when **hold-reclaimed-time** is configured to a value greater than zero), the server will still reclaim leases according to the parameters described in Section 10.3, but the reclaimed leases will be held in the database (rather than removed) for a specified amount of time. When the client returns, the server will first verify whether there are any reclaimed leases associated with this client and will re-assign them if possible. However, it is important to note that any reclaimed lease may be assigned to another client if that client specifically asks for it. Therefore, lease affinity does not guarantee that the reclaimed lease will be available for the client who used it before; it merely increases the chances for the client to be assigned the same lease. If the lease pool is small (this mostly applies to DHCPv4 for which address space is small), there is an increased likelihood that the expired lease will be assigned to another client.

Consider the following configuration:

```
"Dhcp4": {
  ...

  "expired-leases-processing": {
    "reclaim-timer-wait-time": 3,
    "hold-reclaimed-time": 1800,
    "flush-reclaimed-timer-wait-time": 5
  },
  ...
}
```

The **hold-reclaim-time** specifies how many seconds after an expiration a reclaimed lease should be held in the database for re-assignment to the same client. In the example given above, reclaimed leases will be held for 30 minutes (1800s) after their expiration. During this time, the server will likely be able to re-assign the same lease to the returning client, unless another client requests this lease and the server assigns it.

The server must periodically remove reclaimed leases for which the time indicated by **hold-reclaim-time** has elapsed. The **flush-reclaimed-timer-wait-time** parameter controls how often the server removes such leases. In the example provided above, the server will initiate removal of such leases 5 seconds after the previous removal attempt was completed. Setting this value to 0 disables lease affinity, in which case leases will be removed from the lease database when they are reclaimed. If lease affinity is enabled, it is recommended that **hold-reclaim-time** be set to a value significantly higher than the **reclaim-timer-wait-time**, as timely removal of expired-reclaimed leases is less critical than the removal process, which may impact server responsiveness.

There is no guarantee that lease affinity will work every time. If a server is running out of addresses, it will reassign expired addresses to new clients. Also, clients can request specific addresses and the server will try to honor such a request if possible. If you want to ensure a client keeps its address, even after periods of inactivity, consider using host reservations or leases with very long lifetimes.

10.5 Reclaiming Expired Leases with Command

The *leases-reclaim* command can be used to trigger lease reclamation at any time. Please consult the Section 17.3.6 section for details about using this command.



Chapter 11

Congestion Handling in DHCPv4 and DHCPv6

11.1 What is Congestion?

Congestion occurs when servers are subjected to client queries faster than they can be processed. As a result, the servers begin accumulating a backlog of pending queries. The longer the high rate of traffic continues the farther behind the servers fall. Depending on the client implementations, those that fail to get leases either give up or simply continue to retry forever. In the former case, the server may eventually recover, but the latter case is a vicious cycle from which the server is unable to escape.

In a well-planned deployment, the number and capacity of servers is matched to the maximum client loads expected. As long as capacity is matched to load, congestion does not occur. If the load is routinely too heavy, then the deployment needs to be re-evaluated. Congestion typically occurs when there is a network event that causes overly large numbers of clients to simultaneously need leases, such as recovery after a network outage.

The goal of congestion handling is to help servers mitigate the peak in traffic by fulfilling as many of the most relevant requests as possible until the congestion subsides.

Prior to Kea 1.5, `kea-dhcp4` and `kea-dhcp6` read inbound packets directly from the interface sockets in the main application thread, which meant that packets waiting to be processed were held in socket buffers themselves. Once these buffers filled, any new packets were discarded. Under swamped conditions, the servers ended up processing client packets that were no longer relevant, or worse were redundant. In other words, the packets waiting in the FIFO socket buffers became increasingly stale.

11.2 Configuring Congestion Handling

Kea 1.5 introduced the Congestion Handling feature. Congestion handling offers the ability to configure the server to use a separate thread to read packets from the interface socket buffers. As the thread reads packets from the buffers, they are added to an internal packet queue, and the server's main application thread processes packets from this queue rather than from the socket buffers. By structuring it this way, a configurable layer has been introduced which can make decisions on which packets to process, how to store them, and the order in which they are processed by the server.

The default packet queue implementation for both `kea-dhcp4` and `kea-dhcp6` is a simple ring buffer. Once it reaches capacity, new packets get added to the back of the queue by discarding packets from the front of the queue. Rather than always discarding the newest packets, Kea now always discards the oldest packets. The capacity of the buffer, i.e the maximum number of packets the buffer can contain, is configurable. A reasonable starting point would be to match the capacity to the number of leases per second your installation of Kea can handle. Please note that this figure varies widely depending on the specifics of your deployment.

As there is no one algorithm that will best handle the dynamics of all sites, and because over time new approaches will evolve, the packet queue is implemented as a plug-in, which can be replaced by a custom queue implementation via a hook library. This should make it straightforward for interested parties to experiment with their own solutions. (Developers can refer to `isc::dhcp::PacketQueue` and `isc::dhcp::PacketQueueMgr`, described in the Kea Developer's Guide).

Packet queue behavior is configured in both `kea-dhcp4` and `kea-dhcp6` servers through an optional, top-level, configuration element, `'dhcp-queue-control'`. Omitting this element disables packet queueing:



```
"dhcp-queue-control": {  
  "enable-queue": true|false,  
  "queue-type": "queue type",  
  "capacity" : n  
}
```

where:

- **enable-queue** true|false. Enables or disables packet queuing. When true, the server processes packets from the packet queue, which is filled by a separate thread. When false, the server processes packets directly from the socket buffers in the main thread. It is disabled by default.
- **queue-type** name of the queue implementation to use. This value exists so that custom implementations can be registered (via hook library) and then selected. There is a default packet queue implementation that is pre-registered during server start up: "kea-ring4" for kea-dhcp4 and "kea-ring6" for kea-dhcp6.
- **capacity** = n [packets]. This is the maximum number of packets the queue can hold before packets are discarded. The optimal value for this is extremely site-dependent. The default value is 500 for both kea-ring4 and kea-ring6.

The following example enables the default packet queue for kea-dhcp4, with a queue capacity of 250 packets:

```
"Dhcp4":  
{  
  ...  
  "dhcp-queue-control": {  
    "enable-queue": true,  
    "queue-type": "kea-ring4",  
    "capacity" : 250  
  },  
  ...  
}
```

The following example enables the default packet queue for kea-dhcp6, with a queue capacity of 300 packets:

```
"Dhcp6":  
{  
  ...  
  "dhcp-queue-control": {  
    "enable-queue": true,  
    "queue-type": "kea-ring6",  
    "capacity" : 300  
  },  
  ...  
}
```



Chapter 12

The DHCP-DDNS Server

12.1 Overview

The DHCP-DDNS Server (kea-dhcp-ddns, known informally as D2) conducts the client side of the Dynamic DNS protocol (DDNS, defined in [RFC 2136](#)) on behalf of the DHCPv4 and DHCPv6 servers (kea-dhcp4 and kea-dhcp6 respectively). The DHCP servers construct DDNS update requests, known as NameChangeRequests (NCRs), based on DHCP lease change events and then post them to D2. D2 attempts to match each request to the appropriate DNS server(s) and carries out the necessary conversation with those servers to update the DNS data.

12.1.1 DNS Server Selection

In order to match a request to the appropriate DNS servers, D2 must have a catalog of servers from which to select. In fact, D2 has two such catalogs, one for forward DNS and one for reverse DNS; these catalogs are referred to as DDNS Domain Lists. Each list consists of one or more named DDNS Domains. Further, each DDNS Domain has a list of one or more DNS servers that publish the DNS data for that domain.

When conducting forward domain matching, D2 compares the fully-qualified domain name (FQDN) in the request against the name of each Forward DDNS Domain in its catalog. The domain whose name matches the longest portion of the FQDN is considered the best match. For example, if the FQDN is "myhost.sample.example.com.", and there are two forward domains in the catalog, "sample.example.com." and "example.com.", the former is regarded as the best match. In some cases, it may not be possible to find a suitable match. Given the same two forward domains there would be no match for the FQDN, "bogus.net", so the request would be rejected. Finally, if there are no Forward DDNS Domains defined, D2 simply disregards the forward update portion of requests.

When conducting reverse domain matching, D2 constructs a reverse FQDN from the lease address in the request and compares that against the name of each Reverse DDNS Domain. Again, the domain whose name matches the longest portion of the FQDN is considered the best match. For instance, if the lease address is "172.16.1.40" and there are two reverse domains in the catalog, "1.16.172.in-addr.arpa." and "16.172.in-addr.arpa", the former is the best match. As with forward matching, it may not find a suitable match. Given the same two domains, there would be no match for the lease address, "192.168.1.50", and the request would be rejected. Finally, if there are no Reverse DDNS Domains defined, D2 will simply disregard the reverse update portion of requests.

12.1.2 Conflict Resolution

D2 implements the conflict resolution strategy prescribed by [RFC 4703](#). Conflict resolution is intended to prevent different clients from mapping to the same FQDN at the same time. To make this possible, the RFC requires that forward DNS entries for a given FQDN must be accompanied by a DHCID resource record (RR). This record contains a client identifier that uniquely identifies the client to whom the name belongs. Furthermore, any DNS updater that wishes to update or remove existing forward entries for an FQDN may only do so if their client matches that of the DHCID RR.



In other words, the DHCID RR maps an FQDN to the client to whom it belongs, and thereafter changes to that mapping should only be done by or at the behest of that client.

Currently, conflict detection is always performed.

12.1.3 Dual-Stack Environments

[RFC 4703, section 5.2](#), describes issues that may arise with dual-stack clients. These are clients that wish to have both IPv4 and IPv6 mappings for the same FQDN. For this to work properly, the clients are required to embed their IPv6 DUID within their IPv4 client identifier option as described in [RFC 4703](#). In this way, DNS updates for both IPv4 and IPv6 can be managed under the same DHCID RR. Support for this does not yet exist in Kea.

12.2 Starting and Stopping the DHCP-DDNS Server

kea-dhcp-ddns is the Kea DHCP-DDNS server and, due to the nature of DDNS, it runs alongside either the DHCPv4 or DHCPv6 component (or both). Like other parts of Kea, it is a separate binary that can be run on its own or through **keactrl** (see [Chapter 6](#)). In normal operation, controlling **kea-dhcp-ddns** with **keactrl** is recommended; however, it is also possible to run the DHCP-DDNS server directly. It accepts the following command-line switches:

- **-c file** - specifies the configuration file. This is the only mandatory switch.
- **-d** - specifies whether the server logging should be switched to debug/verbose mode. In verbose mode, the logging severity and debuglevel specified in the configuration file are ignored and "debug" severity and the maximum debuglevel (99) are assumed. The flag is convenient for temporarily switching the server into maximum verbosity, e.g. when debugging.
- **-v** - displays the Kea version and exits.
- **-W** - displays the Kea configuration report and exits. The report is a copy of the `config.report` file produced by `./configure`; it is embedded in the executable binary.
- **-t file** specifies the configuration file to be tested. Kea-dhcp-ddns will attempt to load it and will conduct sanity checks. Note that certain checks are possible only while running the actual server. The actual status is reported with an exit code (0 = configuration looks ok, 1 = error encountered). Kea prints out log messages to standard output and errors to standard error when testing the configuration.

The `config.report` may also be accessed more directly, via the following command. The binary **path** may be found in the `install` directory or in the `.libs` subdirectory in the source tree. For example: `kea/src/bin/d2/.libs/kea-dhcp-ddns`.

```
strings path/kea-dhcp-ddns | sed -n 's/;;; //p'
```

Upon startup the module will load its configuration and begin listening for NCRs based on that configuration.

During startup the server will attempt to create a PID file of the form: `[runstatedir]/[conf name].kea-dhcp-ddns.pid` where:

- **runstatedir**: The value as passed into the build configure script; it defaults to `"/usr/local/var/run"`. Note that this value may be overridden at runtime by setting the environment variable `KEA_PIDFILE_DIR`. This is intended primarily for testing purposes.
- **conf name**: The configuration file name used to start the server, minus all preceding paths and the file extension. For example, given a pathname of `"/usr/local/etc/kea/myconf.txt"`, the portion used would be `"myconf"`.

If the file already exists and contains the PID of a live process, the server will issue a `DHCP_DDNS_ALREADY_RUNNING` log message and exit. It is possible, though unlikely, that the file is a remnant of a system crash and the process to which the PID belongs is unrelated to Kea. In such a case it is necessary to manually delete the PID file.



12.3 Configuring the DHCP-DDNS Server

Before starting `kea-dhcp-ddns` module for the first time, a configuration file must be created. The following default configuration is a template that can be customized to your requirements.

```
"DhcpDdns": {
  "ip-address": "127.0.0.1",
  "port": 53001,
  "dns-server-timeout": 100,
  "ncr-protocol": "UDP",
  "ncr-format": "JSON",
  "tsig-keys": [ ],
  "forward-ddns": {
    "ddns-domains": [ ]
  },
  "reverse-ddns": {
    "ddns-domains": [ ]
  }
}
```

The configuration can be divided into the following sections, each of which is described below:

- *Global Server Parameters* - values which control connectivity and global server behavior.
- *Control Socket* - defines the Control Socket type and name.
- *TSIG Key Info* - defines the TSIG keys used for secure traffic with DNS servers.
- *Forward DDNS* - defines the catalog of Forward DDNS Domains.
- *Reverse DDNS* - defines the catalog of Forward DDNS Domains.

12.3.1 Global Server Parameters

- **ip-address** - IP address on which D2 listens for requests. The default is the local loopback interface at address 127.0.0.1. You may specify either an IPv4 or IPv6 address.
- **port** - Port on which D2 listens for requests. The default value is 53001.
- **dns-server-timeout** - Maximum amount of time, in milliseconds, that D2 will wait for a response from a DNS server to a single DNS update message.
- **ncr-protocol** - Socket protocol to use when sending requests to D2. Currently only UDP is supported.
- **ncr-format** - Packet format to use when sending requests to D2. Currently only JSON format is supported.

D2 must listen for change requests on a known address and port. By default it listens at 127.0.0.1 on port 53001. The following example illustrates how to change D2's global parameters so it will listen at 192.168.1.10 port 900:

```
"DhcpDdns": {
  "ip-address": "192.168.1.10",
  "port": 900,
  ...
}
```



Warning

It is possible for a malicious attacker to send bogus NameChangeRequests to the DHCP-DDNS server. Addresses other than the IPv4 or IPv6 loopback addresses (127.0.0.1 or ::1) should only be used for testing purposes, but note that local users may still communicate with the DHCP-DDNS server.

**Note**

If the ip-address and port are changed, the corresponding values in the DHCP servers' "dhcp-ddns" configuration section must be changed.

12.3.2 Management API for the D2 Server

The management API allows the issuing of specific management commands, such as configuration retrieval or shutdown. For more details, see Chapter 17. Currently the only supported communication channel type is UNIX stream socket. By default there are no sockets open. To instruct Kea to open a socket, the following entry in the configuration file can be used:

```
"DhcpDdns": {
  "control-socket": {
    "socket-type": "unix",
    "socket-name": "/path/to/the/unix/socket"
  },
  ...
}
```

The length of the path specified by the **socket-name** parameter is restricted by the maximum length for the unix socket name on your operating system, i.e. the size of the **sun_path** field in the **sockaddr_un** structure, decreased by 1. This value varies on different operating systems between 91 and 107 characters. Typical values are 107 on Linux and 103 on FreeBSD.

Communication over control channel is conducted using JSON structures. See the Control Channel section in the Kea Developer's Guide for more details.

The D2 server supports the following operational commands:

- build-report
- config-get
- config-reload
- config-set
- config-test
- config-write
- list-commands
- shutdown
- version-get

12.3.3 TSIG Key List

A DDNS protocol exchange can be conducted with or without TSIG (defined in [RFC 2845](#)). This configuration section allows the administrator to define the set of TSIG keys that may be used in such exchanges.

To use TSIG when updating entries in a DNS Domain, a key must be defined in the TSIG Key List and referenced by name in that domain's configuration entry. When D2 matches a change request to a domain, it checks whether the domain has a TSIG key associated with it. If so, D2 will use that key to sign DNS update messages sent to and verify responses received from the domain's DNS server(s). For each TSIG key required by the DNS servers that D2 will be working with, there must be a corresponding TSIG key in the TSIG Key list.

As one might gather from the name, the `tsig-key` section of the D2 configuration lists the TSIG keys. Each entry describes a TSIG key used by one or more DNS servers to authenticate requests and sign responses. Every entry in the list has three parameters:



- **name** - a unique text label used to identify this key within the list. This value is used to specify which key (if any) should be used when updating a specific domain. As long as the name is unique its content is arbitrary, although for clarity and ease of maintenance it is recommended that it match the name used on the DNS server(s). This field cannot be blank.
- **algorithm** - specifies which hashing algorithm should be used with this key. This value must specify the same algorithm used for the key on the DNS server(s). The supported algorithms are listed below:
 - **HMAC-MD5**
 - **HMAC-SHA1**
 - **HMAC-SHA224**
 - **HMAC-SHA256**
 - **HMAC-SHA384**
 - **HMAC-SHA512**

This value is not case-sensitive.

- **digest-bits** - is used to specify the minimum truncated length in bits. The default value 0 means truncation is forbidden; non-zero values must be an integral number of octets, and be greater than both 80 and half of the full length. (Note that in BIND 9 this parameter is appended after a dash to the algorithm name.)
- **secret** - is used to specify the shared secret key code for this key. This value is case-sensitive and must exactly match the value specified on the DNS server(s). It is a base64-encoded text value.

As an example, suppose that a domain D2 will be updating is maintained by a BIND 9 DNS server, which requires dynamic updates to be secured with TSIG. Suppose further that the entry for the TSIG key in BIND 9's named.conf file looks like this:

```
:
key "key.four.example.com." {
    algorithm hmac-sha224;
    secret "bZEG7Ow8OgAUPfLWV3aAUQ==";
};
:
```

By default, the TSIG Key list is empty:

```
"DhcpDdns": {
    "tsig-keys": [ ],
    ...
}
```

We must extend the list with a new key:

```
"DhcpDdns": {
    "tsig-keys": [
        {
            "name": "key.four.example.com.",
            "algorithm": "HMAC-SHA224",
            "secret": "bZEG7Ow8OgAUPfLWV3aAUQ=="
        }
    ],
    ...
}
```

These steps would be repeated for each TSIG key needed. Note that the same TSIG key can be used with more than one domain.



12.3.4 Forward DDNS

The Forward DDNS section is used to configure D2's forward update behavior. Currently it contains a single parameter, the catalog of Forward DDNS Domains, which is a list of structures.

```
"DhcpDdns": {
  "forward-ddns": {
    "ddns-domains": [ ]
  },
  ...
}
```

By default, this list is empty, which will cause the server to ignore the forward update portions of requests.

12.3.4.1 Adding Forward DDNS Domains

A Forward DDNS Domain maps a forward DNS zone to a set of DNS servers which maintain the forward DNS data (i.e. name-to-address mapping) for that zone. Each zone served needs one Forward DDNS Domain. It may very well be that some or all of the zones are maintained by the same servers, but you will still need one DDNS Domain per zone. Remember that matching a request to the appropriate server(s) is done by zone and a DDNS Domain only defines a single zone.

This section describes how to add Forward DDNS Domains; repeat these steps for each Forward DDNS Domain desired. Each Forward DDNS Domain has the following parameters:

- **name** - the fully qualified domain name (or zone) that this DDNS Domain can update. This value is compared against the request FQDN during forward matching. It must be unique within the catalog.
- **key-name** - if TSIG is used with this domain's servers, this value should be the name of the key from within the TSIG Key List. If the value is blank (the default), TSIG will not be used in DDNS conversations with this domain's servers.
- **dns-servers** - a list of one or more DNS servers which can conduct the server side of the DDNS protocol for this domain. The servers are used in a first-to-last preference; in other words, when D2 begins to process a request for this domain, it will pick the first server in this list and attempt to communicate with it. If that attempt fails, it will move to next one in the list and so on until either it achieves success or the list is exhausted.

To create a new Forward DDNS Domain, add a new domain element and set its parameters:

```
"DhcpDdns": {
  "forward-ddns": {
    "ddns-domains": [
      {
        "name": "other.example.com.",
        "key-name": "",
        "dns-servers": [
        ]
      }
    ]
  }
}
```

It is possible to add a domain without any servers; however, if that domain matches a request, the request will fail. To make the domain useful, we must add at least one DNS server to it.

12.3.4.1.1 Adding Forward DNS Servers

This section describes how to add DNS servers to a Forward DDNS Domain. Repeat these instructions as needed for all the servers in each domain.

Forward DNS Server entries represent actual DNS servers which support the server side of the DDNS protocol. Each Forward DNS Server has the following parameters:



- **hostname** - the resolvable host name of the DNS server; this parameter is not yet implemented.
- **ip-address** - the IP address at which the server listens for DDNS requests. This may be either an IPv4 or an IPv6 address.
- **port** - the port on which the server listens for DDNS requests. It defaults to the standard DNS service port of 53.

To create a new forward DNS Server, one must add a new server element to the domain and fill in its parameters. If, for example, the service is running at "172.88.99.10", then set the forward DNS Server as follows:

```
"DhcpDdns": {
  "forward-ddns": {
    "ddns-domains": [
      {
        "name": "other.example.com.",
        "key-name": "",
        "dns-servers": [
          {
            "hostname": "",
            "ip-address": "172.88.99.10",
            "port": 53
          }
        ]
      }
    ]
  }
}
```

Note

Since "hostname" is not yet supported, the parameter "ip-address" must be set to the address of the DNS server.

12.3.5 Reverse DDNS

The Reverse DDNS section is used to configure D2's reverse update behavior, and the concepts are the same as for the forward DDNS section. Currently it contains a single parameter, the catalog of Reverse DDNS Domains, which is a list of structures.

```
"DhcpDdns": {
  "reverse-ddns": {
    "ddns-domains": [ ]
  }
  ...
}
```

By default, this list is empty, which will cause the server to ignore the reverse update portions of requests.

12.3.5.1 Adding Reverse DDNS Domains

A Reverse DDNS Domain maps a reverse DNS zone to a set of DNS servers which maintain the reverse DNS data (address-to-name mapping) for that zone. Each zone served needs one Reverse DDNS Domain. It may very well be that some or all of the zones are maintained by the same servers, but you will still need one DDNS Domain entry for each zone. Remember that matching a request to the appropriate server(s) is done by zone and a DDNS Domain only defines a single zone.

This section describes how to add Reverse DDNS Domains; repeat these steps for each Reverse DDNS Domain desired. Each Reverse DDNS Domain has the following parameters:

- **name** - the fully qualified reverse zone that this DDNS Domain can update. This is the value used during reverse matching, which will compare it with a reversed version of the request's lease address. The zone name should follow the appropriate standards; for example, to support the IPv4 subnet 172.16.1, the name should be "1.16.172.in-addr.arpa.". Similarly, to support an IPv6 subnet of 2001:db8:1, the name should be "1.0.0.8.B.D.0.1.0.0.2.ip6.arpa." Whatever the name, it must be unique within the catalog.



- **key-name** - if TSIG should be used with this domain's servers, this value should be the name of that key from the TSIG Key List. If the value is blank (the default), TSIG will not be used in DDNS conversations with this domain's servers. Currently this value is not used as TSIG has not been implemented.
- **dns-servers** - a list of one or more DNS servers which can conduct the server side of the DDNS protocol for this domain. Currently, the servers are used in a first-to-last preference; in other words, when D2 begins to process a request for this domain, it will pick the first server in this list and attempt to communicate with it. If that attempt fails, it will move to the next one in the list and so on until either it achieves success or the list is exhausted.

To create a new Reverse DDNS Domain, one must add a new domain element and set its parameters. For example, to support subnet 2001:db8:1::, the following configuration could be used:

```
"DhcpDdns": {
  "reverse-ddns": {
    "ddns-domains": [
      {
        "name": "1.0.0.0.8.B.D.0.1.0.0.2.ip6.arpa.",
        "key-name": "",
        "dns-servers": [
        ]
      }
    ]
  }
}
```

It is possible to add a domain without any servers; however, if that domain matches a request, the request will fail. To make the domain useful, you must add at least one DNS server to it.

12.3.5.1.1 Adding Reverse DNS Servers

This section describes how to add DNS servers to a Reverse DDNS Domain. Repeat these instructions as needed for all the servers in each domain.

Reverse DNS Server entries represent actual DNS servers which support the server side of the DDNS protocol. Each Reverse DNS Server has the following parameters:

- **hostname** - the resolvable host name of the DNS server; this value is currently ignored.
- **ip-address** - the IP address at which the server listens for DDNS requests.
- **port** - the port on which the server listens for DDNS requests. It defaults to the standard DNS service port of 53.

To create a new reverse DNS Server, one must first add a new server element to the domain and fill in its parameters. If, for example, the service is running at "172.88.99.10", then set it as follows:

```
"DhcpDdns": {
  "reverse-ddns": {
    "ddns-domains": [
      {
        "name": "1.0.0.0.8.B.D.0.1.0.0.2.ip6.arpa.",
        "key-name": "",
        "dns-servers": [
          {
            "hostname": "",
            "ip-address": "172.88.99.10",
            "port": 53
          }
        ]
      }
    ]
  }
}
```

**Note**

Since "hostname" is not yet supported, the parameter "ip-address" must be set to the address of the DNS server.

12.3.6 User Contexts in DDNS

Note

User contexts were designed for hook libraries, which are not yet supported for DHCP-DDNS server configuration.

User contexts can store arbitrary data as long as it has valid JSON syntax and its top level element is a map (i.e. the data must be enclosed in curly brackets).

User contexts can be specified on global scope, ddns domain, dns server, tsig key, and loggers. One other useful usage is the ability to store comments or descriptions; the parser translates a "comment" entry into a user context with the entry, which allows a comment to be attached inside the configuration itself.

12.3.7 Example DHCP-DDNS Server Configuration

This section provides a sample DHCP-DDNS server configuration, based on a small example network. Let's suppose our example network has three domains, each with their own subnet.

Domain	Subnet	Forward DNS Servers	Reverse DNS Servers
four.example.com	192.0.2.0/24	172.16.1.5, 172.16.2.5	172.16.1.5, 172.16.2.5
six.example.com	2001:db8:1::/64	3001:1::50	3001:1::51
example.com	192.0.0.0/16	172.16.2.5	172.16.2.5

Table 12.1: Our Example Network

We need to construct three Forward DDNS Domains: As discussed earlier, FQDN-to-domain matching is based on the longest

#	DDNS Domain Name	DNS Servers
1.	four.example.com.	172.16.1.5, 172.16.2.5
2.	six.example.com.	3001:1::50
3.	example.com.	172.16.2.5

Table 12.2: Forward DDNS Domains Needed

match. The FQDN, "myhost.four.example.com.", will match the first domain ("four.example.com") while "admin.example.com." will match the third domain ("example.com"). The FQDN, "other.example.net.", will fail to match any domain and is rejected.

The following example configuration specifies the Forward DDNS Domains.

```
"DhcpDdns": {
  "comment": "example configuration: forward part",
  "forward-ddns": {
    "ddns-domains": [
      {
        "name": "four.example.com.",
        "key-name": "",
        "dns-servers": [
          { "ip-address": "172.16.1.5" },
          { "ip-address": "172.16.2.5" }
        ]
      }
    ]
  }
},
```



```
{
  "name": "six.example.com.",
  "key-name": "",
  "dns-servers": [
    { "ip-address": "2001:db8::1" }
  ]
},
{
  "name": "example.com.",
  "key-name": "",
  "dns-servers": [
    { "ip-address": "172.16.2.5" }
  ],
  "user-context": { "backup": false }
},
]
}
}</b>
```

Similarly, we need to construct the three Reverse DDNS Domains: An address of "192.0.2.150" will match the first domain,

#	DDNS Domain Name	DNS Servers
1.	2.0.192.in-addr.arpa.	172.16.1.5, 172.16.2.5
2.	1.0.0.0.8.d.b.0.1.0.0.2.ip6.arpa.	3001:1::50
3.	0.182.in-addr.arpa.	172.16.2.5

Table 12.3: Reverse DDNS Domains Needed

"2001:db8:1::10" will match the second domain, and "192.0.50.77" the third domain.

These Reverse DDNS Domains are specified as follows:

```
"DhcpDdns": {
  "comment": "example configuration: reverse part",
  "reverse-ddns": {
    "ddns-domains": [
      {
        "name": "2.0.192.in-addr.arpa.",
        "key-name": "",
        "dns-servers": [
          { "ip-address": "172.16.1.5" },
          { "ip-address": "172.16.2.5" }
        ]
      }
    ]
  }
  {
    "name": "1.0.0.0.8.B.D.0.1.0.0.2.ip6.arpa.",
    "key-name": "",
    "dns-servers": [
      { "ip-address": "2001:db8::1" }
    ]
  }
  {
    "name": "0.192.in-addr.arpa.",
    "key-name": "",
    "dns-servers": [
      { "ip-address": "172.16.2.5" }
    ]
  }
]
}
```



}

12.4 DHCP-DDNS Server Limitations

The following are the current limitations of the DHCP-DDNS Server.

- Requests received from the DHCP servers are placed in a queue until they are processed. Currently, all queued requests are lost when the server shuts down.



Chapter 13

The LFC Process

13.1 Overview

kea-lfc is a service process that removes redundant information from the files used to provide persistent storage for the memfile database backend. This service is written to run as a standalone process.

While **kea-lfc** can be started externally, there is usually no need to do this. **kea-lfc** is run on a periodic basis by the Kea DHCP servers.

The process operates on a set of files, using them to receive input and output of the lease entries and to indicate what stage the process is in, in the event of an interruption. Currently the caller must supply names for all of the files.

13.2 Command-Line Options

kea-lfc is run as follows:

```
kea-lfc [-4 | -6] -c config-file -p pid-file -x previous-file -i copy-file -o output-file - ←  
f finish-file
```

The argument **-4** or **-6** selects the protocol version of the lease files.

The **-c** argument specifies the configuration file. This is required, but is not currently used by the process.

The **-p** argument specifies the PID file. When the **kea-lfc** process starts, it attempts to determine whether another instance of the process is already running by examining the pid file. If one is already running, the new process is terminated; if one is not running, Kea writes its pid into the pid file.

The other filenames specify where the **kea-lfc** process should look for input, write its output, and perform its bookkeeping:

- **previous** — When **kea-lfc** starts, this is the result of any previous run of **kea-lfc**. When **kea-lfc** finishes, it is the result of this run. If **kea-lfc** is interrupted before completing, this file may not exist.
- **input** — Before the DHCP server invokes **kea-lfc**, it will move the current lease file here and then call **kea-lfc** with this file.
- **output** — This is the temporary file where **kea-lfc** writes the leases. Upon completion of writing this file, it will be moved to the finish file (see below).
- **finish** — This is another temporary file **kea-lfc** uses for bookkeeping. When **kea-lfc** completes writing the outputfile, it moves it to this file name. After **kea-lfc** finishes deleting the other files (previous and input), it moves this file to the previous lease file. By moving the files in this fashion, the **kea-lfc** and the DHCP server processes can determine the correct file to use even if one of the processes was interrupted before completing its task.

There are several additional arguments, mostly for debugging purposes. **-d** sets the logging level to debug. **-v** and **-V** print out version stamps, with **-V** providing a longer form. **-h** prints out the usage string.



Chapter 14

Client Classification

14.1 Client Classification Overview

In certain cases it is useful to differentiate between different types of clients and treat them accordingly. Common reasons include:

- The clients represent different pieces of topology, e.g. a cable modem is not the same as the clients behind that modem.
- The clients have different behavior, e.g. a smart phone behaves differently from a laptop.
- The clients require different values for some options, e.g. a docsis3.0 cable modem requires different settings from a docsis2.0 cable modem.

To make management easier, different clients can be grouped into a client class to receive common options.

An incoming packet can be associated with a client class in several ways:

- Implicitly, using a vendor class option or another builtin condition.
- Using an expression which evaluates to true.
- Using static host reservations, a shared network, a subnet, etc.
- Using a hook.

It is envisaged that client classification will be used for changing the behavior of almost any part of the DHCP message processing. There are currently five mechanisms that take advantage of client classification: subnet selection, pool selection, definition of DHCPv4 private (codes 224-254) and code 43 options, assignment of different options, and, for DHCPv4 cable modems, the setting of specific options for use with the TFTP server address and the boot file field.

The classification process is conducted in several steps:

1. The ALL class is associated with the incoming packet.
2. Vendor class options are processed.
3. Classes with matching expressions and not marked for later ("on request" or depending on the KNOWN/UNKNOWN builtin classes) evaluation are processed in the order they are defined in the configuration; the boolean expression is evaluated and, if it returns true ("match"), the incoming packet is associated with the class.
4. If a private or code 43 DHCPv4 option is received, it is decoded following its client class or global (or, for option 43, last resort) definition.
5. When the incoming packet belongs the special class, "DROP", it is dropped and an informational message is logged with the packet information.



6. A subnet is chosen, possibly based on the class information when some subnets are reserved. More precisely: when choosing a subnet, the server iterates over all of the subnets that are feasible given the information found in the packet (client address, relay address, etc). It uses the first subnet it finds that either doesn't have a class associated with it, or has a class which matches one of the packet's classes.
7. The server looks for host reservations. If an identifier from the incoming packet matches a host reservation in the subnet or shared network, the packet is associated with the KNOWN class and all classes of the host reservation. If a reservation is not found, the packet is assigned to the UNKNOWN class.
8. Classes with matching expressions - directly or indirectly using the KNOWN/UNKNOWN builtin classes and not marked for later ("on request") evaluation - are processed in the order they are defined in the configuration; the boolean expression is evaluated and, if it returns true ("match"), the incoming packet is associated with the class. After a subnet is selected, the server determines whether there is a reservation for a given client. Therefore, it is not possible to use KNOWN/UNKNOWN classes to select a shared network or a subnet, nor to make the DROP class dependent of KNOWN/UNKNOWN classes.
9. If needed, addresses and prefixes from pools are assigned, possibly based on the class information when some pools are reserved for class members.
10. Classes marked as "required" are evaluated in the order in which they are listed: first the shared network, then the subnet, and finally the pools that assigned resources belong to.
11. Options are assigned, again possibly based on the class information in the order that classes were associated with the incoming packet. For DHCPv4 private and code 43 options, this includes class local option definitions.

Note

Client classes in Kea follow the order in which they are specified in the configuration (vs. alphabetical order). Required classes follow the order in which they are required.

When determining which options to include in the response, the server examines the union of options from all of the assigned classes. If two or more classes include the same option, the value from the first class examined is used; classes are examined in the order they were associated, so ALL is always the first class and matching required classes are last.

As an example, imagine that an incoming packet matches two classes. Class "foo" defines values for an NTP server (option 42 in DHCPv4) and an SMTP server (option 69 in DHCPv4), while class "bar" defines values for an NTP server and a POP3 server (option 70 in DHCPv4). The server examines the three options - NTP, SMTP, and POP3 - and returns any that the client requested. As the NTP server was defined twice, the server chooses only one of the values for the reply; the class from which the value is obtained is unspecified.

Note

Care should be taken with client classification as it is easy for clients that do not meet any class criteria to be denied service altogether.

14.2 Builtin Client Classes

Some classes are builtin, so they do not need to be defined. The main example uses Vendor Class information: The server checks whether an incoming DHCPv4 packet includes the vendor class identifier option (60) or an incoming DHCPv6 packet includes the vendor class option (16). If it does, the content of that option is prepended with "VENDOR_CLASS_" and the result is interpreted as a class. For example, modern cable modems send this option with value "docsis3.0", so the packet belongs to class "VENDOR_CLASS_docsis3.0".

The "HA_" prefix is used by the High Availability hooks library to designate certain servers to process DHCP packets as a result of load balancing. The class name is constructed by prepending the "HA_" prefix to the name of the server which should process the DHCP packet. This server will use an appropriate pool or subnet to allocate IP addresses (and/or prefixes), based on the assigned client classes. The details can be found in [Section 15.4.9](#).



Other examples are: the ALL class, which all incoming packets belong to, and the KNOWN class, assigned when host reservations exist for a particular client. By convention, builtin classes' names begin with all capital letters.

Currently recognized builtin class names are ALL, KNOWN and UNKNOWN, and prefixes `VENDOR_CLASS_`, `HA_`, `AFTER_`, and `EXTERNAL_`. Although the `AFTER_` prefix is a provision for an as-yet-unwritten hook, the `EXTERNAL_` prefix can be freely used; builtin classes are implicitly defined so they never raise warnings if they do not appear in the configuration.

14.3 Using Expressions in Classification

The expression portion of a classification definition contains operators and values. All values are currently strings; operators take a string or strings and return another string. When all the operations have completed, the result should be a value of "true" or "false". The packet belongs to the class (and the class name is added to the list of classes) if the result is "true". Expressions are written in standard format and can be nested.

Expressions are pre-processed during the parsing of the configuration file and converted to an internal representation. This allows certain types of errors to be caught and logged during parsing. Examples of these errors include an incorrect number or type of argument to an operator. The evaluation code also checks for this class of error and generally throws an exception, though this should not occur in a normally functioning system.

Other issues, for example the starting position of a substring being outside of the substring or an option not existing in the packet, result in the operator returning an empty string.

Dependencies between classes are also checked. For instance, forward dependencies are rejected when the configuration is parsed; an expression can only depend on already-defined classes (including builtin classes) which are evaluated in a previous or the same evaluation phase. This does not apply to the KNOWN or UNKNOWN classes.

Notes:

- Hexadecimal strings are converted into a string as expected. The starting "0X" or "0x" is removed, and if the string is an odd number of characters a "0" is prepended to it.
- IP addresses are converted into strings of length 4 or 16. IPv4, IPv6, and IPv4-embedded IPv6 (e.g., IPv4-mapped IPv6) addresses are supported.
- Integers in an expression are converted to 32-bit unsigned integers and are represented as four-byte strings; for example, 123 is represented as 0x0000007b. All expressions that return numeric values use 32-bit unsigned integers, even if the field in the packet is smaller. In general, it is easier to use decimal notation to represent integers, but it is also possible to use hexadecimal notation. When writing an integer in hexadecimal, care should be taken to make sure the value is represented as 32 bits, e.g. use 0x00000001 instead of 0x1 or 0x01. Also, make sure the value is specified in network order, e.g. 1 is represented as 0x00000001.
- "option[code].hex" extracts the value of the option with the code "code" from the incoming packet. If the packet doesn't contain the option, it returns an empty string. The string is presented as a byte string of the option payload, without the type code or length fields.
- "option[code].exists" checks whether an option with the code "code" is present in the incoming packet. It can be used with empty options.
- "member('foobar')" checks whether the packet belongs to the client class "foobar". To avoid dependency loops, the configuration file parser verifies whether client classes were already defined or are builtin, i.e., beginning by "VENDOR_CLASS_", "AFTER_" (for the to come "after" hook) and "EXTERNAL_" or equal to "ALL", "KNOWN", "UNKNOWN" etc.
"known" and "unknown" are short hands for "member('KNOWN')" and "not member('KNOWN')". Note the evaluation of any expression using directly or indirectly the "KNOWN" class is deferred after the host reservation lookup (i.e. when the "KNOWN" or "UNKNOWN" partition is determined).
- "relay4[code].hex" attempts to extract the value of the sub-option "code" from the option inserted as the DHCPv4 Relay Agent Information (82) option. If the packet doesn't contain a RAI option, or the RAI option doesn't contain the requested sub-option, the expression returns an empty string. The string is presented as a byte string of the option payload without the type code or length fields. This expression is allowed in DHCPv4 only.



Name	Example expression	Example value
String literal	'example'	'example'
Hexadecimal string literal	0x5a7d	'Z}'
IP address literal	10.0.0.1	0x0a000001
Integer literal	123	'123'
Integer literal	123	'123'
Binary content of the option	option[123].hex	'(content of the option)'
Option existence	option[123].exists	'true'
Client class membership	member('foobar')	'true'
Known client	known	member('KNOWN')
Unknown client	unknown	not member('KNOWN')
DHCPv4 relay agent sub-option	relay4[123].hex	'(content of the RAI sub-option)'
DHCPv6 Relay Options	relay6[nest].option[code].hex	(value of the option)
DHCPv6 Relay Peer Address	relay6[nest].peeraddr	2001:DB8::1
DHCPv6 Relay Link Address	relay6[nest].linkaddr	2001:DB8::1
Interface name of packet	pkt.iface	eth0
Source address of packet	pkt.src	10.1.2.3
Destination address of packet	pkt.dst	10.1.2.3
Length of packet	pkt.len	513
Hardware address in DHCPv4 packet	pkt4.mac	0x010203040506
Hardware length in DHCPv4 packet	pkt4.hlen	6
Hardware type in DHCPv4 packet	pkt4.htype	6
ciaddr field in DHCPv4 packet	pkt4.ciaddr	192.0.2.1
giaddr field in DHCPv4 packet	pkt4.giaddr	192.0.2.1
yiaddr field in DHCPv4 packet	pkt4.yiaddr	192.0.2.1
siaddr field in DHCPv4 packet	pkt4.siaddr	192.0.2.1
Message type in DHCPv4 packet	pkt4.msgtype	1
Transaction ID (xid) in DHCPv4 packet	pkt4.transid	12345
Message type in DHCPv6 packet	pkt6.msgtype	1
Transaction ID in DHCPv6 packet	pkt6.transid	12345
Vendor option existence (any vendor)	vendor[*].exists	true
Vendor option existence (specific vendor)	vendor[4491].exists	true
Enterprise-id from vendor option	vendor.enterprise	4491
Vendor sub-option existence	vendor[4491].option[1].exists	true
Vendor sub-option content	vendor[4491].option[1].hex	docsis3.0
Vendor class option existence (any vendor)	vendor-class[*].exists	true
Vendor class option existence (specific vendor)	vendor-class[4491].exists	true
Enterprise-id from vendor class option	vendor-class.enterprise	4491
First data chunk from vendor class option	vendor-class[4491].data	docsis3.0
Specific data chunk from vendor class option	vendor-class[4491].data[3]	docsis3.0

Table 14.1: List of Classification Values



- "relay4" shares the same representation types as "option", for instance "relay4[code].exists" is supported.
- "relay6[nest]" allows access to the encapsulations used by any DHCPv6 relays that forwarded the packet. The "nest" level specifies the relay from which to extract the information, with a value of 0 indicating the relay closest to the DHCPv6 server. Negative values allow to specify relays counted from the DHCPv6 client, -1 indicating the relay closest to the client. In general negative "nest" level is the same as the number of relays + "nest" level. If the requested encapsulation doesn't exist an empty string "" is returned. This expression is allowed in DHCPv6 only.
- "relay6[nest].option[code]" shares the same representation types as "option", for instance "relay6[nest].option[code].exists" is supported.
- Expressions starting with "pkt4" can be used only in DHCPv4. They allow access to DHCPv4 message fields.
- "pkt6" refers to information from the client request. To access any information from an intermediate relay use "relay6". "pkt6.msgtype" and "pkt6.transid" output a 4 byte binary string for the message type or transaction id. For example the message type SOLICIT will be "0x00000001" or simply 1 as in "pkt6.msgtype == 1".
- Vendor option means Vendor-Identifying Vendor-specific Information option in DHCPv4 (code 125, see [Section 4 of RFC 3925](#)) and Vendor-specific Information Option in DHCPv6 (code 17, defined in [Section 21.17 of RFC 8415](#)). Vendor class option means Vendor-Identifying Vendor Class Option in DHCPv4 (code 124, see [Section 3 of RFC 3925](#)) in DHCPv4 and Class Option in DHCPv6 (code 16, see [Section 21.16 of RFC 8415](#)). Vendor options may have sub-options that are referenced by their codes. Vendor class options do not have sub-options, but rather data chunks, which are referenced by index value. Index 0 means the first data chunk, Index 1 is for the second data chunk (if present), etc.
- In the vendor and vendor-class constructs Asterisk (*) or 0 can be used to specify a wildcard enterprise-id value, i.e. it will match any enterprise-id value.
- Vendor Class Identifier (option 60 in DHCPv4) can be accessed using option[60] expression.
- [RFC 3925](#) and [RFC 8415](#) allow for multiple instances of vendor options to appear in a single message. The client classification code currently examines the first instance if more than one appear. For vendor.enterprise and vendor.class.enterprise expressions, the value from the first instance is returned. Please submit a feature request on Kea website if you need support for multiple instances.

Name	Example	Description
Equal	'foo' == 'bar'	Compare the two values and return "true" or "false"
Not	not ('foo' == 'bar')	Logical negation
And	('foo' == 'bar') and ('bar' == 'foo')	Logical and
Or	('foo' == 'bar') or ('bar' == 'foo')	Logical or
Substring	substring('foobar',0,3)	Return the requested substring
Concat	concat('foo','bar')	Return the concatenation of the strings
Ifelse	ifelse('foo' == 'bar','us','them')	Return the branch value according to the condition
Hexstring	hexstring('foo', '-')	Converts the value to a hexadecimal string, e.g. 0a:1b:2c:3e

Table 14.2: List of Classification Expressions

14.3.1 Logical operators

The Not, And and Or logical operators are the common operators. Not has the highest precedence and Or the lowest. And and Or are (left) associative, parentheses around a logical expression can be used to enforce a specific grouping, for instance in "A and (B or C)" (without parentheses "A and B or C" means "(A and B) or C").



14.3.2 Substring

The substring operator "substring(value, start, length)" accepts both positive and negative values for the starting position and the length. For "start", a value of 0 is the first byte in the string while -1 is the last byte. If the starting point is outside of the original string an empty string is returned. "length" is the number of bytes to extract. A negative number means to count towards the beginning of the string but doesn't include the byte pointed to by "start". The special value "all" means to return all bytes from start to the end of the string. If length is longer than the remaining portion of the string then the entire remaining portion is returned. Some examples may be helpful:

```
substring('foobar', 0, 6) == 'foobar'
substring('foobar', 3, 3) == 'bar'
substring('foobar', 3, all) == 'bar'
substring('foobar', 1, 4) == 'ooba'
substring('foobar', -5, 4) == 'ooba'
substring('foobar', -1, -3) == 'oba'
substring('foobar', 4, -2) == 'ob'
substring('foobar', 10, 2) == ''
```

14.3.3 Concat

The concat function "concat(string1, string2)" returns the concatenation of its two arguments. For instance:

```
concat('foo', 'bar') == 'foobar'
```

14.3.4 Ifelse

The ifelse function "ifelse(cond, iftrue, ifelse)" returns the "iftrue" or "ifelse" branch value following the boolean condition "cond". For instance:

```
ifelse(option[230].exists, option[230].hex, 'none')
```

14.3.5 Hexstring

The hexstring function "hexstring(binary, separator)" returns the binary value as its hexadecimal string representation: pairs of hexadecimal digits separated by the separator, e.g ':', '-', "" (empty separator).

```
hexstring(pkt4.mac, ':')
```

Note

The expression for each class is executed on each packet received. If the expressions are overly complex, the time taken to execute them may impact the performance of the server. If you need complex or time consuming expressions you should write a **hook** to perform the necessary work.

14.4 Configuring Classes

A class contains five items: a name, a test expression, option data, option definition and only-if-required flag. The name must exist and must be unique amongst all classes. The test expression, option data and definition, and only-if-required flag are optional.

The test expression is a string containing the logical expression used to determine membership in the class. The entire expression is in double quotes.



The option data is a list which defines any options that should be assigned to members of this class.

The option definition is for DHCPv4 option 43 (Section 8.2.13 and DHCPv4 private options (Section 8.2.12).

Usually the test expression is evaluated before subnet selection but in some cases it is useful to evaluate it later when the subnet, shared-network or pools are known but output option processing not yet done. The only-if-required flag, false by default, allows to perform the evaluation of the test expression only when it was required, i.e. in a require-client-classes list of the selected subnet, shared-network or pool.

The require-client-classes list which is valid for shared-network, subnet and pool scope specifies the classes which are evaluated in the second pass before output option processing. The list is built in the reversed precedence order of option data, i.e. an option data in a subnet takes precedence on one in a shared-network but required class in a subnet is added after one in a shared-network. The mechanism is related to the only-if-required flag but it is not mandatory that the flag was set to true.

In the following example the class named "Client_foo" is defined. It is comprised of all clients whose client ids (option 61) start with the string "foo". Members of this class will be given 192.0.2.1 and 192.0.2.2 as their domain name servers.

```
"Dhcp4": {
  "client-classes": [
    {
      "name": "Client_foo",
      "test": "substring(option[61].hex,0,3) == 'foo'",
      "option-data": [
        {
          "name": "domain-name-servers",
          "code": 6,
          "space": "dhcp4",
          "csv-format": true,
          "data": "192.0.2.1, 192.0.2.2"
        }
      ]
    }
  ],
  ...
},
...
```

This example shows a client class being defined for use by the DHCPv6 server. In it the class named "Client_enterprise" is defined. It is comprised of all clients who's client identifiers start with the given hex string (which would indicate a DUID based on an enterprise id of 0xAABBCCDD). Members of this class will be given an 2001:db8:0::1 and 2001:db8:2::1 as their domain name servers.

```
"Dhcp6": {
  "client-classes": [
    {
      "name": "Client_enterprise",
      "test": "substring(option[1].hex,0,6) == 0x0002AABBCCDD",
      "option-data": [
        {
          "name": "dns-servers",
          "code": 23,
          "space": "dhcp6",
          "csv-format": true,
          "data": "2001:db8:0::1, 2001:db8:2::1"
        }
      ]
    }
  ],
  ...
},
...
```



14.5 Using Static Host Reservations In Classification

Classes can be statically assigned to the clients using techniques described in Section 8.3.6 and Section 9.3.5.

14.6 Configuring Subnets With Class Information

In certain cases it beneficial to restrict access to certain subnets only to clients that belong to a given class, using the "client-class" keyword when defining the subnet.

Let's assume that the server is connected to a network segment that uses the 192.0.2.0/24 prefix. The Administrator of that network has decided that addresses from range 192.0.2.10 to 192.0.2.20 are going to be managed by the DHCP4 server. Only clients belonging to client class Client_foo are allowed to use this subnet. Such a configuration can be achieved in the following way:

```
"Dhcp4": {
  "client-classes": [
    {
      "name": "Client_foo",
      "test": "substring(option[61].hex,0,3) == 'foo'",
      "option-data": [
        {
          "name": "domain-name-servers",
          "code": 6,
          "space": "dhcp4",
          "csv-format": true,
          "data": "192.0.2.1, 192.0.2.2"
        }
      ]
    },
    ...
  ],
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [ { "pool": "192.0.2.10 - 192.0.2.20" } ],
      "client-class": "Client_foo"
    },
    ...
  ],
  ...
}
```

The following example shows restricting access to a DHCPv6 subnet. This configuration will restrict use of the addresses 2001:db8:1::1 to 2001:db8:1::FFFF to members of the "Client_enterprise" class.

```
"Dhcp6": {
  "client-classes": [
    {
      "name": "Client_enterprise",
      "test": "substring(option[1].hex,0,6) == 0x0002AABBCCDD",
      "option-data": [
        {
          "name": "dns-servers",
          "code": 23,
          "space": "dhcp6",
          "csv-format": true,
          "data": "2001:db8:0::1, 2001:db8:2::1"
        }
      ]
    }
  ]
}
```



```
    },
    ...
  ],
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "pools": [ { "pool": "2001:db8:1::-2001:db8:1::ffff" } ],
      "client-class": "Client_enterprise"
    }
  ],
  ...
}
```

14.7 Configuring Pools With Class Information

Similar to subnets in certain cases access to certain address or prefix pools must be restricted to only clients that belong to a given class, using the "client-class" when defining the pool.

Let's assume that the server is connected to a network segment that uses the 192.0.2.0/24 prefix. The Administrator of that network has decided that addresses from range 192.0.2.10 to 192.0.2.20 are going to be managed by the DHCP4 server. Only clients belonging to client class Client_foo are allowed to use this pool. Such a configuration can be achieved in the following way:

```
"Dhcp4": {
  "client-classes": [
    {
      "name": "Client_foo",
      "test": "substring(option[61].hex,0,3) == 'foo'",
      "option-data": [
        {
          "name": "domain-name-servers",
          "code": 6,
          "space": "dhcp4",
          "csv-format": true,
          "data": "192.0.2.1, 192.0.2.2"
        }
      ]
    }
  ],
  ...
},
"subnet4": [
  {
    "subnet": "192.0.2.0/24",
    "pools": [
      {
        "pool": "192.0.2.10 - 192.0.2.20",
        "client-class": "Client_foo"
      }
    ]
  }
],
...
},
},
```

The following example shows restricting access to an address pool. This configuration will restrict use of the addresses 2001:db8:1::1 to 2001:db8:1::FFFF to members of the "Client_enterprise" class.

```
"Dhcp6": {
```



```
"client-classes": [
  {
    "name": "Client_enterprise_",
    "test": "substring(option[1].hex,0,6) == 0x0002AABBCCDD",
    "option-data": [
      {
        "name": "dns-servers",
        "code": 23,
        "space": "dhcp6",
        "csv-format": true,
        "data": "2001:db8:0::1, 2001:db8:2::1"
      }
    ]
  },
  ...
],
"subnet6": [
  {
    "subnet": "2001:db8:1::/64",

    "pools": [
      {
        "pool": "2001:db8:1::-2001:db8:1::ffff",
        "client-class": "Client_foo"
      }
    ]
  },
  ...
],
...
}
```

14.8 Using Classes

Currently classes can be used for two functions. They can supply options to the members of the class and they can be used to choose a subnet from which an address will be assigned to the class member.

When supplying options, options defined as part of the class definition are considered "class globals". They will override any global options that may be defined and in turn will be overridden by any options defined for an individual subnet.

14.9 Classes and Hooks

You may use a hook to classify your packets. This may be useful if the expression would either be complex or time consuming and be easier or better to write as code. Once the hook has added the proper class name to the packet the rest of the classification system will work as normal in choosing a subnet and selecting options. For a description of hooks see [Chapter 15](#), for a description on configuring classes see [Section 14.4](#) and [Section 14.6](#).

14.10 Debugging Expressions

While you are constructing your classification expressions you may find it useful to enable logging see [Chapter 18](#) for a more complete description of the logging facility.

To enable the debug statements in the classification system you will need to set the severity to "DEBUG" and the debug level to at least 55. The specific loggers are "kea-dhcp4.eval" and "kea-dhcp6.eval".



In order to understand the logging statements, one must understand a bit about how expressions are evaluated; for a more complete description refer to the design document at <https://gitlab.isc.org/isc-projects/kea/wikis/design%20documents>. In brief there are two structures used during the evaluation of an expression: a list of tokens which represent the expressions and a value stack which represents the values being manipulated.

The list of tokens is created when the configuration file is processed with most expressions and values being converted to a token. The list is organized in reverse Polish notation. During execution, the list will be traversed in order. As each token is executed it will be able to pop values from the top of the stack and eventually push its result on the top of the stack. Imagine the following expression:

```
"test": "substring(option[61].hex,0,3) == 'foo'",
```

This will result in the following tokens:

```
option, number (0), number (3), substring, text ('foo'), equals
```

In this example the first three tokens will simply push values onto the stack. The substring token will then remove those three values and compute a result that it places on the stack. The text option also places a value on the stack and finally the equals token removes the two tokens on the stack and places its result on the stack.

When debug logging is enabled, each time a token is evaluated it will emit a log message indicating the values of any objects that were popped off of the value stack and any objects that were pushed onto the value stack.

The values will be displayed as either text if the command is known to use text values or hexadecimal if the command either uses binary values or can manipulate either text or binary values. For expressions that pop multiple values off the stack, the values will be displayed in the order they were popped. For most expressions this won't matter but for the concat expression the values are displayed in reverse order from how they are written in the expression.

Let us assume that the following test has been entered into the configuration. This example skips most of the configuration to concentrate on the test.

```
"test": "substring(option[61].hex,0,3) == 'foo'",
```

The logging might then resemble this:

```
2016-05-19 13:35:04.163 DEBUG [kea.eval/44478] EVAL_DEBUG_OPTION Pushing option 61 ←  
with value 0x666F6F626172  
2016-05-19 13:35:04.164 DEBUG [kea.eval/44478] EVAL_DEBUG_STRING Pushing text string ←  
'0'  
2016-05-19 13:35:04.165 DEBUG [kea.eval/44478] EVAL_DEBUG_STRING Pushing text string ←  
'3'  
2016-05-19 13:35:04.166 DEBUG [kea.eval/44478] EVAL_DEBUG_SUBSTRING Popping length ←  
3, start 0, string 0x666F6F626172 pushing result 0x666F6F  
2016-05-19 13:35:04.167 DEBUG [kea.eval/44478] EVAL_DEBUG_STRING Pushing text string ←  
'foo'  
2016-05-19 13:35:04.168 DEBUG [kea.eval/44478] EVAL_DEBUG_EQUAL Popping 0x666F6F and ←  
0x666F6F pushing result 'true'
```

Note

The debug logging may be quite verbose if you have a number of expressions to evaluate. It is intended as an aid in helping you create and debug your expressions. You should plan to disable debug logging when you have your expressions working correctly. You also may wish to include only one set of expressions at a time in the configuration file while debugging them in order to limit the log statements. For example when adding a new set of expressions you might find it more convenient to create a configuration file that only includes the new expressions until you have them working correctly and then add the new set to the main configuration file.



Chapter 15

Hooks Libraries

15.1 Introduction

Although Kea offers a lot of flexibility, there may be cases where its behavior needs customization. To accommodate this possibility, Kea includes the idea of "Hooks". This feature lets Kea load one or more dynamically-linked libraries (known as "hooks libraries") and, at various points in its processing ("hook points"), call functions in them. Those functions perform whatever custom processing is required.

The hooks concept also allows keeping the core Kea code reasonably small by moving features that some, but not all users find useful to external libraries. People who don't need specific functionality simply don't load the libraries.

Hooks libraries are loaded by individual Kea processes, not to Kea as a whole. This means (for example) that it is possible to associate one set of libraries with the DHCP4 server and a different set to the DHCP6 server.

Another point to note is that it is possible for a process to load multiple libraries. When processing reaches a hook point, Kea calls the hooks library functions attached to it. If multiple libraries have attached a function to a given hook point, Kea calls all of them, in the order in which the libraries are specified in the configuration file. The order may be important: consult the documentation of the libraries to see if this is the case.

The next section describes how to configure hooks libraries. If you are interested in writing your own hooks library, information can be found in the [Kea Developer's Guide](#).

Note that some libraries are available under different licenses.

Note that some libraries may require additional dependencies and/or compilation switches to be enabled, e.g. Radius library introduced in Kea 1.4 requires FreeRadius-client library to be present. If `--with-free-radius` option is not specified, the Radius library will not be built.

15.2 Installing Hook packages

Note

The installation procedure has changed in 1.4.0. Kea 1.3.0 and earlier needed special switches passed to configure script to detect the hook libraries. Please see this KB article: <https://kb.isc.org/article/AA-01587>.

Some hook packages are included in the base Kea sources. There is no need to do anything special to compile or install them, they are covered by the usual building and installation procedure. ISC also provides several additional hooks in form of various packages. All of those packages follow the same installation procedure that is similar to base Kea, but has several additional steps. For your convenience, the whole procedure is described here. Please refer to Chapter 3 for general overview.

1. Download the package. You will receive detailed instructions how to get it separately. This will be a file with a name similar to `kea-premium-1.6.0-beta2.tar.gz`. Your name may differ depending on which package you got.



2. If you have the sources for the corresponding version of the open-source Kea package still on your system (from when you installed Kea), skip this step. Otherwise extract the Kea source from the original tarball you downloaded. For example, if you downloaded Kea 1.6.0-beta2., you should have a tarball called `kea-1.6.0-beta2.tar.gz` on your system. Unpack this tarball:

```
$ tar zxvf kea-1.6.0-beta2.tar.gz
```

This will unpack the tarball into the `kea-1.6.0-beta2` subdirectory of your current working directory.

3. Unpack the Kea premium tarball into the directory into which Kea was unpacked. For example, assuming that you followed step 2 and that Kea 1.6.0-beta2 has been unpacked into a `kea-1.6.0-beta2` subdirectory and that the Kea premium tarball is in your current directory, the following steps will unpack the premium tarball into the correct location:

```
$ cd kea-1.6.0-beta2
$ tar xvf ../kea-premium-1.6.0-beta2.tar.gz
```

Note that unpacking the Kea premium package will put the files into a directory named `premium`. Regardless of the name of your package, the directory will always be called `premium`, just its content may vary.

4. Run `autoreconf` tools. This step is necessary to update Kea's build script to include additional directory. If this tool is not already available on your system, you need to install `automake` and `autoconf` tools. To generate configure script, please use:

```
$ autoreconf -i
```

5. Rerun `configure`, using the same configure options as you used when originally building Kea. You can check if `configure` has detected the premium package by inspecting the summary printed when it exits. The first section of the output should look something like:

```
Package:
  Name:          kea
  Version:       1.6.0-beta2
  Extended version: 1.6.0-beta2 (tarball)
  OS Family:     Linux
  Using GNU sed: yes
  Premium package: yes
  Included Hooks: forensic_log flex_id host_cmds
```

The last line indicates which specific hooks were detected. Note that some hooks may require its own dedicated switches, e.g. `radius` hook requires extra switches for `FreeRADIUS`. Please consult later sections of this chapter for details.

6. Rebuild Kea

```
$ make
```

If your machine has multiple CPU cores, interesting option to consider here is `-j X`, where `X` is the number of available cores.

7. Install Kea sources together with hooks:

```
$ sudo make install
```

Note that as part of the installation procedure, the `install` script will eventually venture into `premium/` directory and will install additional hook libraries and associated files.

The installation location of the hooks libraries depends whether you specified `--prefix` parameter to the `configure` script. If you did not, the default location will be `/usr/local/lib/kea/hooks`. You can verify the libraries are installed properly with this command:

```
$ ls -l /usr/local/lib/kea/hooks/*.so
/usr/local/lib/kea/hooks/libdhcp_class_cmds.so
/usr/local/lib/kea/hooks/libdhcp_flex_id.so
/usr/local/lib/kea/hooks/libdhcp_host_cmds.so
/usr/local/lib/kea/hooks/libdhcp_lease_cmds.so
/usr/local/lib/kea/hooks/libdhcp_legal_log.so
/usr/local/lib/kea/hooks/libdhcp_subnet_cmds.so
```

The exact list you see will depend on the packages you have. If you specified directory via `--prefix`, the hooks libraries will be located in `{prefix directory}/lib/kea/hooks`.



15.3 Configuring Hooks Libraries

The hooks libraries for a given process are configured using the **hooks-libraries** keyword in the configuration for that process. (Note that the word "hooks" is plural). The value of the keyword is an array of map structures, each structure corresponding to a hooks library. For example, to set up two hooks libraries for the DHCPv4 server, the configuration would be:

```
"Dhcp4": {
  :
  "hooks-libraries": [
    {
      "library": "/opt/charging.so"
    },
    {
      "library": "/opt/local/notification.so",
      "parameters": {
        "mail": "spam@example.com",
        "floor": 13,
        "debug": false,
        "users": [ "alice", "bob", "charlie" ],
        "languages": {
          "french": "bonjour",
          "klingon": "yl'el"
        }
      }
    }
  ]
  :
}
```

Note

This is a change to the syntax used in Kea 0.9.2 and earlier, where `hooks-libraries` was a list of strings, each string being the name of a library. The change was made in Kea 1.0 to facilitate the specification of library-specific parameters, a capability available in Kea 1.1.0 onwards. Libraries should allow a parameter entry where to put comments as it is done for many configuration scopes with `comment` and `user-context`.

Note

The library reloading behavior has changed in Kea 1.1.0. Libraries are reloaded, even if their list hasn't changed. Kea does that, because the parameters specified for the library (or the files those parameters point to) may have changed.

Libraries may have additional parameters. Those are not mandatory in the sense that there may be libraries that don't require them. However, for specific library there is often specific requirement for specify certain set of parameters. Please consult the documentation for your library for details. In the example above, the first library has no parameters. The second library has five parameters, specifying mail (string parameter), floor (integer parameter), debug (boolean parameter) and even lists (list of strings) and maps (containing strings). Nested parameters could be used if the library supports it. This topic is explained in detail in the Hooks Developer's Guide in the "Configuring Hooks Libraries" section.

Notes:

- The full path to each library should be given.
 - As noted above, order may be important - consult the documentation for each library.
 - An empty list has the same effect as omitting the **hooks-libraries** configuration element all together.
-

**Note**

There is one case where this is not true: if Kea is running with a configuration that contains a **hooks-libraries** item, and that item is removed and the configuration reloaded, the removal will be ignored and the libraries remain loaded. As a workaround, instead of removing the **hooks-libraries** item, change it to an empty list. This will be fixed in a future version of Kea.

At the present time, only the kea-dhcp4 and kea-dhcp6 processes support hooks libraries.

15.4 Available Hooks Libraries

As described above, the hooks functionality provides a way to customize a Kea server without modifying the core code. ISC has chosen to take advantage of this feature to provide functions that may only be useful to a subset of Kea users. To this end ISC has created some hooks libraries; these discussed in the following sections.

Note

Some of these libraries will be available with the base code while others will be shared with organizations supporting development of Kea, possibly as a 'benefit' or 'thank you' for helping to sustain the larger Kea project. If you would like to get access to those libraries, please consider taking out a support contract: this includes professional support, advance security notifications, input into our roadmap planning, and many other benefits, while helping making Kea sustainable in the long term.

The following table provides a list of libraries currently available from ISC. It is important to pay attention to which libraries may be loaded by which Kea processes. It is a common mistake to configure the **kea-ctrl-agent** process to load libraries that should, in fact, be loaded by the **kea-dhcp4** or **kea-dhcp6** processes. If a library from ISC doesn't work as expected, please make sure that it has been loaded by the correct process per the table below.

**Warning**

While the Kea Control Agent includes the "hooks" functionality, (i.e. hooks libraries can be loaded by this process), none of ISC's current hooks libraries should be loaded by the Control Agent.

Name	Availability	Since
user_chk	Kea sources	Kea 0.8
Forensic Logging	Support customers	Kea 1.1.0
Flexible Identifier	Support customers	Kea 1.2.0
Host Commands	Support customers	Kea 1.2.0
Subnet Commands	Support customers	Kea 1.3.0
Lease Commands	Kea sources	Kea 1.3.0
High Availability	Kea sources	Kea 1.4.0
Statistics Commands	Kea sources	Kea 1.4.0
Radius	Support customers	Kea 1.4.0
Host Cache	Support customers	Kea 1.4.0
Class Commands	Support customers	Kea 1.5.0
MySQL Configuration Backend	Kea sources	Kea 1.6.0
Configuration Backend Commands	Support customers	Kea 1.6.0

Table 15.1: List of available hooks libraries

ISC hopes to see more hooks libraries become available as time progresses, both developed internally and externally. Since this list may evolve dynamically, we decided to keep it on a wiki page, available at this link: <https://gitlab.isc.org/isc-projects/-/kea/wikis/Hooks-available>. If you are a developer or are aware of any hooks libraries not listed there, please send a note to the kea-users or kea-dev mailing lists and someone will update it.

The libraries developed by ISC are described in detail in the following sections.



15.4.1 user_chk: Checking User Access

The user_chk library is the first hooks library published by ISC. It attempts to serve several purposes:

- To assign "new" or "unregistered" users to a restricted subnet, while "known" or "registered" users are assigned to unrestricted subnets.
- To allow DHCP response options or vendor option values to be customized based upon user identity.
- To provide a real time record of the user registration activity which can be sampled by an external consumer.
- To serve as a demonstration of various capabilities possible using the hooks interface.

Once loaded, the library allows segregating incoming requests into known and unknown clients. For known clients, the packets are processed mostly as usual, except it is possible to override certain options being sent. That can be done on a per host basis. Clients that are not on the known hosts list will be treated as unknown and will be assigned to the last subnet defined in the configuration file.

As an example of use, this behavior may be used to put unknown users into a separate subnet that leads to a walled garden, where they can only access a registration portal. Once they fill in necessary data, their details are added to the known clients file and they get a proper address after their device is restarted.

Note

This library was developed several years before the host reservation mechanism has become available. Currently host reservation is much more powerful and flexible, but nevertheless the user_chk capability to consult an external source of information about clients and alter Kea's behavior is useful and remains of educational value.

The library reads the /tmp/user_chk_registry.txt file while being loaded and each time an incoming packet is processed. The file is expected to have each line contain a self-contained JSON snippet which must have the following two entries:

- **type**, whose value is "HW_ADDR" for IPv4 users or "DUID" for IPv6 users
- **id**, whose value is either the hardware address or the DUID from the request formatted as a string of hex digits, with or without ":" delimiters.

and may have the zero or more of the following entries:

- **bootfile** whose value is the pathname of the desired file
- **tftp_server** whose value is the hostname or IP address of the desired server

A sample user registry file is shown below:

```
{ "type" : "HW_ADDR", "id" : "0c:0e:0a:01:ff:04", "bootfile" : "/tmp/v4bootfile" }
{ "type" : "HW_ADDR", "id" : "0c:0e:0a:01:ff:06", "tftp_server" : "tftp.v4.example.com" }
{ "type" : "DUID", "id" : "00:01:00:01:19:ef:e6:3b:00:0c:01:02:03:04", "bootfile" : "/tmp/ ↵
v6bootfile" }
{ "type" : "DUID", "id" : "00:01:00:01:19:ef:e6:3b:00:0c:01:02:03:06", "tftp_server" : " ↵
tftp.v6.example.com" }
```

As with any other hooks libraries provided by ISC, internals of the user_chk code are well documented. You can take a look at the [Kea Developer's Guide section dedicated to the user_chk library](#) that discusses how the code works internally. That, together with our general entries in [Hooks Framework section](#) should give you some pointers how to extend this library and perhaps even write your own from scratch.



15.4.2 legal_log: Forensic Logging Hooks

This section describes the forensic log hooks library. This library provides hooks that record a detailed log of lease assignments and renewals into a set of log files. Currently this library is only available to ISC customers with a support contract.

Note

This library may only be loaded by **kea-dhcp4** or **kea-dhcp6** process.

In many legal jurisdictions companies, especially ISPs, must record information about the addresses they have leased to DHCP clients. This library is designed to help with that requirement. If the information that it records is sufficient it may be used directly. If your jurisdiction requires that you save a different set of information you may use it as a template or example and create your own custom logging hooks.

This logging is done as a set of hooks to allow it to be customized to any particular need. Modifying a hooks library is easier and safer than updating the core code. In addition by using the hooks features those users who don't need to log this information can leave it out and avoid any performance penalties.

15.4.2.1 Log File Naming

The names for the log files have the following form:

```
path/base-name.CCYMMDD.txt
```

The "path" and "base-name" are supplied in the configuration as described below see Section [15.4.2.4](#). The next part of the name is the date the log file was started, with four digits for year, two digits for month and two digits for day. The file is rotated on a daily basis.

Note

When running Kea servers for both DHCPv4 and DHCPv6 the log names must be distinct. See the examples in Section [15.4.2.4](#).

15.4.2.2 DHCPv4 Log Entries

For DHCPv4 the library creates entries based on DHCPREQUEST messages and corresponding DHCPv4 leases intercepted by `lease4_select` (for new leases) and `lease4_renew` (for renewed leases) hooks.

An entry is a single string with no embedded end-of-line markers, a prepended timestamp and has the following sections:

```
timestamp address duration device-id {client-info} {relay-info} {user-context}
```

Where:

- `timestamp` - the current date and time the log entry was written in "%Y-%m-%d %H:%M:%S %Z" strftime format ("%Z" is the time zone name).
 - `address` - the leased IPv4 address given out and whether it was assigned or renewed.
 - `duration` - the lease lifetime expressed in days (if present), hours, minutes and seconds. A lease lifetime of 0xFFFFFFFF will be denoted with the text "infinite duration".
 - `device-id` - the client's hardware address shown as numerical type and hex digit string.
 - `client-info` - the DHCP client id option (61) if present, shown as a hex string.
 - `relay-info` - for relayed packets the giaddr and the RAI circuit-id, remote-id and subscriber-id options (option 82 sub options: 1, 2 and 6) if present. The circuit id and remote id are presented as hex strings
-



- user-context - the optional user context associated to the lease.

For instance (line breaks added for readability, they would not be present in the log file).

```
2018-01-06 01:02:03 CET Address: 192.2.1.100 has been renewed for 1 hrs 52 min 15 secs to a ↵
device with hardware address:
hwtype=1 08:00:2b:02:3f:4e, client-id: 17:34:e2:ff:09:92:54 connected via relay at address: ↵
192.2.16.33,
identified by circuit-id: 68:6f:77:64:79 and remote-id: 87:f6:79:77:ef
```

In addition to logging lease activity driven by DHCPv4 client traffic, it also logs entries for the following lease management control channel commands: lease4-add, lease4-update, and lease4-del. Each entry is a single string with no embedded end-of-line markers and they will typically have the following forms:

lease4-add:

```
*timestamp* Administrator added a lease of address: *address* to a device with hardware ↵
address: *device-id*
```

Dependent on the arguments of the add command, it may also include the client-id and duration.

Example:

```
2018-01-06 01:02:03 CET Administrator added a lease of address: 192.0.2.202 to a device ↵
with hardware address:
1a:1b:1c:1d:1e:1f for 1 days 0 hrs 0 mins 0 secs
```

lease4-update:

```
*timestamp* Administrator updated information on the lease of address: *address* to a ↵
device with hardware address: *device-id*
```

Dependent on the arguments of the update command, it may also include the client-id and lease duration.

Example:

```
2018-01-06 01:02:03 CET Administrator updated information on the lease of address: ↵
192.0.2.202 to a device
with hardware address: 1a:1b:1c:1d:1e:1f, client-id: 1234567890
```

lease4-del: Deletes have two forms, one by address and one by identifier and identifier type:

```
*timestamp* Administrator deleted the lease for address: *address*
```

or

```
*timestamp* Administrator deleted a lease for a device identified by: *identifier-type* of ↵
*identifier*
```

Currently only a type of @b hw-address (hardware address) is supported.

Examples:

```
2018-01-06 01:02:03 CET Administrator deleted the lease for address: 192.0.2.202
```

```
2018-01-06 01:02:12 CET Administrator deleted a lease for a device identified by: hw- ↵
address of 1a:1b:1c:1d:1e:1f
```



15.4.2.3 DHCPv6 Log Entries

For DHCPv6 the library creates entries based on lease management actions intercepted by the `lease6_select` (for new leases), `lease6_renew` (for renewed leases) and `lease6_rebind` (for rebound leases).

An entry is a single string with no embedded end-of-line markers, a prepended timestamp and has the following sections:

```
timestamp address duration device-id {relay-info}* {user-context}
```

Where:

- `timestamp` - the current date and time the log entry was written in "%Y-%m-%d %H:%M:%S %Z" strftime format ("%Z" is the time zone name).
- `address` - the leased IPv6 address or prefix given out and whether it was assigned or renewed.
- `duration` - the lease lifetime expressed in days (if present), hours, minutes and seconds. A lease lifetime of 0xFFFFFFFF will be denoted with the text "infinite duration".
- `device-id` - the client's DUID and hardware address (if present).
- `relay-info` - for relayed packets the content of relay agent messages, remote-id (code 37), subscriber-id (code 38) and interface-id (code 18) options if present. Note that interface-id option, if present, identifies the whole interface the relay agent received the message on. This typically translates to a single link in your network, but it depends on your specific network topology. Nevertheless, this is useful information to better scope down the location of the device, so it is being recorded, if present.
- `user-context` - the optional user context associated to the lease.

For instance (line breaks added for readability, they would not be present in the log file).

```
2018-01-06 01:02:03 PST Address:2001:db8:1:: has been assigned for 0 hrs 11 mins 53 secs
to a device with DUID: 17:34:e2:ff:09:92:54 and hardware address: hwtype=1 08:00:2b:02:3f:4 ←
e
(from Raw Socket) connected via relay at address: fe80::abcd for client on link address: ←
3001::1,
hop count: 1, identified by remote-id: 01:02:03:04:0a:0b:0c:0d:0e:0f and subscriber-id: 1a ←
:2b:3c:4d:5e:6f
```

In addition to logging lease activity driven by DHCPv6 client traffic, it also logs entries for the following lease management control channel commands: `lease6-add`, `lease6-update`, and `lease6-del`. Each entry is a single string with no embedded end-of-line markers and they will typically have the following forms:

lease6-add:

```
*timestamp* Administrator added a lease of address: *address* to a device with DUID: *DUID*
```

Dependent on the arguments of the add command, it may also include the hardware address and duration.

Example:

```
2018-01-06 01:02:03 PST Administrator added a lease of address: 2001:db8::3 to a device ←
with DUID:
1a:1b:1c:1d:1e:1f:20:21:22:23:24 for 1 days 0 hrs 0 mins 0 secs
```

lease6-update:

```
*timestamp* Administrator updated information on the lease of address: *address* to a ←
device with DUID: *DUID*
```

Dependent on the arguments of the update command, it may also include the hardware address and lease duration.

Example:



```
2018-01-06 01:02:03 PST Administrator updated information on the lease of address: 2001:db8 ←
::3 to a device with
DUID: 1a:1b:1c:1d:1e:1f:20:21:22:23:24, hardware address: 1a:1b:1c:1d:1e:1f
```

lease6-del: Deletes have two forms, one by address and one by identifier and identifier type:

```
*timestamp* Administrator deleted the lease for address: *address*
```

or

```
*timestamp* Administrator deleted a lease for a device identified by: *identifier-type* of ←
*identifier*
```

Currently only a type of DUID is supported.

Examples:

```
2018-01-06 01:02:03 PST Administrator deleted the lease for address: 2001:db8::3
```

```
2018-01-06 01:02:11 PST Administrator deleted a lease for a device identified by: duid of 1 ←
a:1b:1c:1d:1e:1f:20:21:22:23:24
```

15.4.2.4 Configuring the Forensic Log Hooks

To use this functionality the hook library must be included in the configuration of the desired DHCP server modules. The `legal_log` library is installed alongside the Kea libraries in `[kea-install-dir]/var/lib/kea` where `kea-install-dir` is determined by the `--prefix` option of the configure script. It defaults to `/usr/local`. Assuming the default value then, configuring `kea-dhcp4` to load the `legal_log` library could be done with the following Kea4 configuration:

```
"Dhcp4": {
  "hooks-libraries": [
    {
      "library": "/usr/local/lib/kea/hooks/libdhcp_legal_log.so",
      "parameters": {
        "path": "/var/lib/kea/log",
        "base-name": "kea-forensic4"
      }
    },
    ...
  ]
}
```

To configure it for `kea-dhcp6`, the commands are simply as shown below:

```
"Dhcp6": {
  "hooks-libraries": [
    {
      "library": "/usr/local/lib/kea/hooks/libdhcp_legal_log.so",
      "parameters": {
        "path": "/var/lib/kea/log",
        "base-name": "kea-forensic6"
      }
    },
    ...
  ]
}
```

Two Hook Library parameters are supported:



- `path` - the directory in which the forensic file(s) will be written. The default value is `[prefix]/var/lib/kea`. The directory must exist.
- `base-name` - an arbitrary value which is used in conjunction with the current system date to form the current forensic file name. It defaults to `kea-legal`.

If it is desired to restrict forensic logging to certain subnets, the "legal-logging" boolean parameter can be specified within a user context of these subnets. For example:

```
"Dhcpv4" {
  "subnet4": [
    {
      "subnet": "192.0.2.0/24",
      "pools": [
        {
          "pool": "192.0.2.1 - 192.0.2.200"
        }
      ],
      "user-context": {
        "legal-logging": false
      }
    }
  ]
}
```

disables legal logging for the subnet "192.0.2.0/24". If this parameter is not specified, it defaults to 'true', which enables legal logging for the subnet.

The following example demonstrates how to selectively disable legal logging for an IPv6 subnet.

```
"Dhcpv6": {
  "subnet6": [
    {
      "subnet": "2001:db8:1::/64",
      "pools": [
        {
          "pool": "2001:db8:1::1-2001:db8:1::ffff"
        }
      ],
      "user-context": {
        "legal-logging": false
      }
    }
  ]
}
```

See Section 8.10 and Section 9.15 to learn more about user contexts in Kea configuration.

15.4.2.5 Database backend

Log entries can be inserted into a database when Kea is configured with database backend support: a table named 'logs' is used with a timestamp (timeuuid for Cassandra CQL) generated by the database software and a text log with the same format than for files without the timestamp.

Please refer to Section 4.3.2 for MySQL, to Section 4.3.3 for PostgreSQL or to Section 4.3.4 for Cassandra CQL. The logs table is part of the Kea database schemas.

Configuration parameters are extended by standard lease database parameters as defined in Section 8.2.2.2. The "type" parameter should be "mysql", "postgresql", "cql" or be "logfile". When it is absent or set to "logfile" files are used.

This database feature is experimental and will be likely improved, for instance to add an address / prefix index (currently the only index is the timestamp). No specific tools is provided to operate the database but standard tools are applicable, for instance to dump the logs table from a CQL database:



```
$ echo 'SELECT dateOf(timeuuid), log FROM logs;' | cqlsh -k database-name
system.dateof(timeuuid)      | log
-----+-----
2018-01-06 01:02:03.227000+0000 | Address: 192.2.1.100 has been renewed ...
...
(12 rows)
$
```

15.4.3 flex_id: Flexible Identifiers for Host Reservations

This section describes a hook application dedicated to generate flexible identifiers for host reservation. Kea software provides a way to handle host reservations that include addresses, prefixes, options, client classes and other features. The reservation can be based on hardware address, DUID, circuit-id or client-id in DHCPv4 and using hardware address or DUID in DHCPv6. However, there are sometimes scenarios where the reservation is more complex, e.g. uses other options that mentioned above, uses part of specific options or perhaps even a combination of several options and fields to uniquely identify a client. Those scenarios are addressed by the Flexible Identifiers hook application.

Currently this library is only available to ISC customers with a support contract.

Note

This library may only be loaded by **kea-dhcp4** or **kea-dhcp6** process.

The library allows for defining an expression, using notation initially used for client classification only. See Section 14.3 for detailed description of the syntax available. One notable difference is that for client classification the expression currently has to evaluate to either true or false, while the flexible identifier expression is expected to evaluate to a string that will be used as identifier. It is a valid case for the expression to evaluate to empty string (e.g. in cases where a client does not sent specific options). This expression is then evaluated for each incoming packet. This evaluation generates an identifier that is used to identify the client. In particular, there may be host reservations that are tied to specific values of the flexible identifier.

The library can be loaded in similar way as other hook libraries. It takes a mandatory parameter **identifier-expression** and optional boolean parameter **replace-client-id**:

```
"Dhcp6": {
  "hooks-libraries": [
    {
      "library": "/path/libdhcp_flex_id.so",
      "parameters": {
        "identifier-expression": "expression",
        "replace-client-id": false
      }
    },
    ...
  ]
}
```

The flexible identifier library supports both DHCPv4 and DHCPv6.

EXAMPLE: Let's consider a case of an IPv6 network that has an independent interface for each of the connected customers. Customers are able to plug in whatever device they want, so any type of identifier (e.g. a client-id) is unreliable. Therefore the operator may decide to use an option inserted by a relay agent to differentiate between clients. In this particular deployment, the operator verified that the interface-id is unique for each customer facing interface. Therefore it is suitable for usage as reservation. However, only the first 6 bytes of the interface-id are interesting, because remaining bytes are either randomly changed or not unique between devices. Therefore the customer decided to use first 6 bytes of the interface-id option inserted by the relay agent. After adding "flex-id" host-reservation-identifiers goal can be achieved by using the following configuration:



```
"Dhcp6": {
  "subnet6": [{ ... , // subnet definition starts here
  "reservations": [
    "flex-id": "'port1234'", // value of the first 8 bytes of the interface-id
    "ip-addresses": [ "2001:db8::1" ]
  ],
}], // end of subnet definitions
"host-reservation-identifiers": ["duid", "flex-id"], // add "flex-id" to reservation ←
  identifiers
"hooks-libraries": [
  {
    "library": "/path/libdhcp_flex_id.so",
    "parameters": {
      "identifier-expression": "substring(relay6[0].option[18].hex, 0, 8) "
    }
  },
  ...
]
}
```

NOTE: Care should be taken when adjusting the expression. If the expression changes, then all the flex-id values may change, possibly rendering all reservations based on flex-id unusable until they're manually updated. Therefore it is strongly recommended to start with the expression and a handful reservations, adjust the expression as needed and only after it was confirmed the expression does exactly what is expected out of it go forward with host reservations on any broader scale.

flex-id values in host reservations can be specified in two ways. First, they can be expressed as hex string, e.g. bar string can be represented as 626174. Alternatively, it can be expressed as quoted value (using double and single quotes), e.g. "bar". The former is more convenient for printable characters, while hex string values are more convenient for non-printable characters and does not require the use of the **hexstring** operator.

```
"Dhcp6": {
  "subnet6": [{ ... , // subnet definition starts here
  "reservations": [
    "flex-id": "01:02:03:04:05:06", // value of the first 8 bytes of the interface-id
    "ip-addresses": [ "2001:db8::1" ]
  ],
}], // end of subnet definitions
"host-reservation-identifiers": ["duid", "flex-id"], // add "flex-id" to reservation ←
  identifiers
"hooks-libraries": [
  {
    "library": "/path/libdhcp_flex_id.so",
    "parameters": {
      "identifier-expression": "vendor[4491].option[1026].hex"
    }
  },
  ...
]
}
```

When "replace-client-id" is set to false (which is the default setting), the flex-id hook library uses evaluated flexible identifier solely for identifying host reservations, i.e. searching for reservations within a database. This is a functional equivalent of other identifiers, similar to hardware address or circuit-id. However, this mode of operation has an implication that if a client device is replaced, it may cause a conflict between an existing lease (allocated for old device) and the new lease being allocated for the new device. The conflict arises because the same flexible identifier is computed for the replaced device and the server will try to allocate the same lease. The mismatch between client identifiers sent by new device and old device causes the server to refuse this new allocation until the old lease expires. A manifestation of this problem is dependant on specific expression used as flexible identifier and is likely to appear if you only use options and other parameters that are identifying where the device is connected (e.g. circuit-id), rather than the device identification itself (e.g. MAC address).



The flex-id library offers a way to overcome the problem with lease conflicts by dynamically replacing client identifier (or DUID in DHCPv6 case) with a value derived from flexible identifier. The server processes the client's query as if flexible identifier was sent in the client identifier (or DUID) option. This guarantees that returning client (for which the same flexible identifier is evaluated) will be assigned the same lease despite the client identifier and/or MAC address change.

The following is a stub configuration that enables this behavior:

```
"Dhcp4": {
  "hooks-libraries": [
    {
      "library": "/path/libdhcp_flex_id.so",
      "parameters": {
        "identifier-expression": "expression",
        "replace-client-id": true
      }
    },
    ...
  ]
}
```

In the DHCPv4 case, the value derived from the flexible identifier is formed by prepending 1 byte with a value of zero to flexible identifier. In the IPv6 case, it is formed by prepending two zero bytes before the flexible identifier.

Note that for this mechanism to take effect, the DHCPv4 server must be configured to respect the client identifier option value during lease allocation, i.e. "match-client-id" must be set to true. See Section 8.2.21 for details. No additional settings are required for DHCPv6.

If "replace-client-id" option is set to true, the value of "echo-client-id" parameter (that governs whether to send back a client-id option or not) is ignored.

The Section 15.4.5 section describes commands used to retrieve, update and delete leases using various identifiers, e.g. "hw-address", "client-id". The lease_cmds library doesn't natively support querying for leases by flexible identifier. However, when "replace-client-id" is set to true, it makes it possible to query for leases using a value derived from the flexible identifier. In the DHCPv4 case, the query will look similar to this:

```
{
  "command": "lease4-get",
  "arguments": {
    "identifier-type": "client-id",
    "identifier": "00:54:64:45:66",
    "subnet-id": 44
  }
}
```

where hexadecimal value of "54:64:45:66" is a flexible identifier computed for the client.

In the DHCPv6 case, the corresponding query will look similar to this:

```
{
  "command": "lease6-get",
  "arguments": {
    "identifier-type": "duid",
    "identifier": "00:00:54:64:45:66",
    "subnet-id": 10
  }
}
```

15.4.4 host_cmds: Host Commands

This section describes a hook application that offers a number of new commands used to query and manipulate host reservations. Kea provides a way to store host reservations in a database. In many larger deployments it is useful to be able to manage that



information while the server is running. This library provides management commands for adding, querying and deleting host reservations in a safe way without restarting the server. In particular, it validates the parameters, so an attempt to insert incorrect data e.g. add a host with conflicting identifier in the same subnet will be rejected. Those commands are exposed via command channel (JSON over unix sockets) and Control Agent (JSON over RESTful interface). Additional commands and capabilities related to host reservations will be added in the future.

Currently this library is only available to ISC customers with a support contract.

Note

This library may only be loaded by **kea-dhcp4** or **kea-dhcp6** process.

Currently five commands are supported: reservation-add (which adds new host reservation), reservation-get (which returns existing reservation if specified criteria are matched), reservation-get-all (which returns all reservations in a specified subnet), reservation-get-page (variant of reservation-get-all which returns all reservations in a specified subnet by pages) and reservation-del (which attempts to delete a reservation matching specified criteria). To use commands that change the reservation information (currently these are reservation-add and reservation-del, but this rule applies to other commands that may be implemented in the future), hosts database must be specified (see hosts-databases description in Section 8.2.3.1 and Section 9.2.3.1) and it must not operate in read-only mode. If the hosts-databases are not specified or are running in read-only mode, the host_cmds library will load, but any attempts to use reservation-add or reservation-del will fail.

Additional host reservation commands are planned in the future. For a description of envisaged commands, see [Control API Requirements](#) document.

All commands are using JSON syntax. They can be issued either using control channel (see Chapter 17) or via Control Agent (see Chapter 7).

The library can be loaded in similar way as other hook libraries. It does not take any parameters. It supports both DHCPv4 and DHCPv6 servers.

```
"Dhcp6": {
  "hooks-libraries": [
    {
      "library": "/path/libdhcp_host_cmds.so"
    }
    ...
  ]
}
```

15.4.4.1 The subnet-id parameter

Prior to diving into the individual commands, it is worth discussing the parameter, **subnet-id**. Currently it is mandatory for all of the commands supplied by this library. Prior to Kea 1.5.0, reservations had to belong to specific subnet. Beginning with Kea 1.5.0, reservations may now be specified globally. In other words, they are not specific to any subnet. When reservations are supplied via the configuration file, the ID of the containing subnet (or lack thereof) is implicit in the configuration structure. However, when managing reservations using the host commands, it is necessary to explicitly identify the scope to which the reservation belongs. This is done via the **subnet-id** parameter. For global reservations, use a value of zero (0). For reservations scoped to a specific subnet, use that subnet's ID.

15.4.4.2 reservation-add command

reservation-add allows for the insertion of a new host. It takes a set of arguments that vary depending on the nature of the host reservation. Any parameters allowed in the configuration file that pertain to host reservation are permitted here. For details regarding IPv4 reservations, see Section 8.3 and Section 9.3. The **subnet-id** is mandatory. Use a value of zero (0) to add a global reservation, or the id of the subnet to which the reservation should be added. An example command can be as simple as:



```
{
  "command": "reservation-add",
  "arguments": {
    "reservation": {
      "subnet-id": 1,
      "hw-address": "1a:1b:1c:1d:1e:1f",
      "ip-address": "192.0.2.202"
    }
  }
}
```

but can also take many more parameters, for example:

```
{
  "command": "reservation-add",
  "arguments": {
    "reservation":
    {
      "subnet-id":1,
      "client-id": "01:0a:0b:0c:0d:0e:0f",
      "ip-address": "192.0.2.205",
      "next-server": "192.0.2.1",
      "server-hostname": "ha19000",
      "boot-file-name": "/dev/null",
      "option-data": [
        {
          "name": "domain-name-servers",
          "data": "10.1.1.202,10.1.1.203"
        }
      ],
      "client-classes": [ "special_snowflake", "office" ]
    }
  }
}
```

Here is an example of complex IPv6 reservation:

```
{
  "command": "reservation-add",
  "arguments": {
    "reservation":
    {
      "subnet-id":1,
      "duid": "01:02:03:04:05:06:07:08:09:0A",
      "ip-addresses": [ "2001:db8:1:cafe::1" ],
      "prefixes": [ "2001:db8:2:abcd::/64" ],
      "hostname": "foo.example.com",
      "option-data": [
        {
          "name": "vendor-opts",
          "data": "4491"
        },
        {
          "name": "tftp-servers",
          "space": "vendor-4491",
          "data": "3000:1::234"
        }
      ]
    }
  }
}
```



The command returns a status that indicates either a success (result 0) or a failure (result 1). Failed command always includes text parameter that explains the cause of failure. Example results:

```
{ "result": 0, "text": "Host added." }
```

Example failure:

```
{ "result": 1, "text": "Mandatory 'subnet-id' parameter missing." }
```

As **reservation-add** is expected to store the host, hosts-databases parameter must be specified in your configuration and databases must not run in read-only mode. In the future versions it will be possible to modify the reservations read from a configuration file. Please contact ISC if you are interested in this functionality.

15.4.4.3 reservation-get command

reservation-get can be used to query the host database and retrieve existing reservations. There are two types of parameters this command supports: (subnet-id, address) or (subnet-id, identifier-type, identifier). The first type of query is used when the address (either IPv4 or IPv6) is known, but the details of the reservation aren't. One common use case of this type of query is to find out whether a given address is reserved or not. The second query uses identifiers. For maximum flexibility, Kea stores the host identifying information as a pair of values: type and the actual identifier. Currently supported identifiers are "hw-address", "duid", "circuit-id", "client-id" and "flex-id", but additional types may be added in the future. If any new identifier types are defined in the future, reservation-get command will support them automatically. The **subnet-id** is mandatory. Use a value of zero (0) to fetch a global reservation, or the id of the subnet to which the reservation belongs.

An example command for getting a host reservation by (subnet-id, address) pair looks as follows:

```
{
  "command": "reservation-get",
  "arguments": {
    "subnet-id": 1,
    "ip-address": "192.0.2.202"
  }
}
```

An example query by (subnet-id, identifier-type, identifier) looks as follows:

```
{
  "command": "reservation-get",
  "arguments": {
    "subnet-id": 4,
    "identifier-type": "hw-address",
    "identifier": "01:02:03:04:05:06"
  }
}
```

reservation-get typically returns result 0 when the query was conducted properly. In particular, 0 is returned when the host was not found. If the query was successful a number of host parameters will be returned. An example of a query that did not find the host looks as follows:

```
{ "result": 0, "text": "Host not found." }
```

An example result returned when the host was found:

```
{
  "arguments": {
    "boot-file-name": "bootfile.efi",
    "client-classes": [
    ],
    "hostname": "somehost.example.org",
  }
}
```



```
"hw-address": "01:02:03:04:05:06",
"ip-address": "192.0.2.100",
"next-server": "192.0.0.2",
"option-data": [

],
"server-hostname": "server-hostname.example.org"
},
"result": 0,
"text": "Host found."
}
```

An example result returned when the query was malformed:

```
{ "result": 1, "text": "No 'ip-address' provided and 'identifier-type'
                        is either missing or not a string." }
```

15.4.4.4 reservation-get-all command

reservation-get-all can be used to query the host database and retrieve all reservations in a specified subnet. This command uses parameters providing the mandatory `subnet-id`. Global host reservations can be retrieved by using `subnet-id` value of zero (0).

For instance for retrieving host reservations for the subnet 1:

```
{
  "command": "reservation-get-all",
  "arguments": {
    "subnet-id": 1
  }
}
```

returns found some IPv4 hosts:

```
{
  "arguments": {
    "hosts": [
      {
        "boot-file-name": "bootfile.efi",
        "client-classes": [ ],
        "hostname": "somehost.example.org",
        "hw-address": "01:02:03:04:05:06",
        "ip-address": "192.0.2.100",
        "next-server": "192.0.0.2",
        "option-data": [ ],
        "server-hostname": "server-hostname.example.org"
      },
      ...
      {
        "boot-file-name": "bootfile.efi",
        "client-classes": [ ],
        "hostname": "otherhost.example.org",
        "hw-address": "01:02:03:04:05:ff",
        "ip-address": "192.0.2.200",
        "next-server": "192.0.0.2",
        "option-data": [ ],
        "server-hostname": "server-hostname.example.org"
      }
    ]
  },
  "result": 0,
  "text": "72 IPv4 host(s) found."
}
```



The response returned by **reservation-get-all** can be very long. The DHCP server does not handle DHCP traffic when preparing a response to reservation-get-all. If there are many reservations in a subnet, this may be disruptive. Use with caution. For larger deployments, please consider using **reservation-get-page** instead (see Section 15.4.4.5).

For a reference, see Section A.109 [402].

15.4.4.5 reservation-get-page command

reservation-get-page can be used to query the host database and retrieve all reservations in a specified subnet by pages. This command uses parameters providing the mandatory subnet-id. Use a value of zero (0) to fetch global reservations. The second mandatory parameter is the page size limit. Optional source-index and from host id, both defaulting to 0, are used to chain page queries.

The usage of from and source-index parameters requires additional explanation. For the first call those parameters should not be specified (or specified as zeros). For any follow up calls they should be set to the values returned in previous calls in a next map holding from and source-index values. The subsequent calls should be issued until all reservations are returned. The end is reached once the returned list is empty, count is 0, no next map is present and result status 3 (empty) is returned.

Note

The from and source-index parameters are reflecting internal state of the search. There is no need to understand what they represent, it's simply a value that is supposed to be copied from one response to the next query. However, if you are curious, from field represents a 64 bits representation of host identifier used by a host backend. The source-index represents internal representation of multiple host backends: 0 is used to represent hosts defined in a configuration file, 1 represents the first database backend. In some uncommon cases there may be more than one database backend configured, so potentially there may be 2. In any case, Kea will iterate over all backends configured.

For instance for retrieving host reservations for the subnet 1 requesting the first page can be done by:

```
{
  "command": "reservation-get-page",
  "arguments": {
    "subnet-id": 1,
    "limit": 10
  }
}
```

Since this is the first call, source-index and from should not be specified. They will default to their zero default values.

Some hosts are returned with informations to get the next page:

```
{
  "arguments": {
    "count": 72,
    "hosts": [
      {
        "boot-file-name": "bootfile.efi",
        "client-classes": [ ],
        "hostname": "somehost.example.org",
        "hw-address": "01:02:03:04:05:06",
        "ip-address": "192.0.2.100",
        "next-server": "192.0.0.2",
        "option-data": [ ],
        "server-hostname": "server-hostname.example.org"
      },
      ...
      {
        "boot-file-name": "bootfile.efi",
        "client-classes": [ ],
        "hostname": "otherhost.example.org",
```



```
        "hw-address": "01:02:03:04:05:ff",
        "ip-address": "192.0.2.200",
        "next-server": "192.0.0.2",
        "option-data": [ ],
        "server-hostname": "server-hostname.example.org"
    },
    ],
    "next": {
        "from": 1234567,
        "source-index": 1
    }
},
"result": 0,
"text": "72 IPv4 host(s) found."
}
```

Note that `from` and `source-index` fields were specified in the response in the next map. Those two must be copied to the next command, so Kea continues from the place where the last command finished. To get the next page the following command can be sent:

```
{
  "command": "reservation-get-page",
  "arguments": {
    "subnet-id": 1,
    "source-index": 1,
    "from": 1234567,
    "limit": 10
  }
}
```

. The response will contain a list of hosts with updated `source-index` and `from` fields. Continue calling the command until you get the last page. Its response will look like this:

```
{
  "arguments": {
    "count": 0,
    "hosts": [ ],
  },
  "result": 3,
  "0 IPv4 host(s) found."
}
```

This command is more complex than **reservation-get-all**, but lets users retrieve larger host reservations lists by smaller chunks. For small deployments with few reservations, is it easier to use **reservation-get-all** (see Section 15.4.4.4).

For a reference, see Section A.110 [403].

Note

Currently **reservation-get-page** is not supported by the Cassandra host backend.

15.4.4.6 reservation-del command

reservation-del can be used to delete a reservation from the host database. There are two types of parameters this command supports: (subnet-id, address) or (subnet-id, identifier-type, identifier). The first type of query is used when the address (either IPv4 or IPv6) is known, but the details of the reservation aren't. One common use case of this type of query is to remove a reservation (e.g. you want a specific address to no longer be reserved). The second query uses identifiers. For maximum flexibility, Kea stores the host identifying information as a pair of values: type and the actual identifier. Currently supported identifiers are "hw-address", "duid", "circuit-id", "client-id" and "flex-id", but additional types may be added in the future. If



any new identifier types are defined in the future, reservation-get command will support them automatically. The **subnet-id** is mandatory. Use a value of zero (0) to delete a global reservation, or the id of the subnet from which the reservation should be deleted.

An example command for deleting a host reservation by (subnet-id, address) pair looks as follows:

```
{
  "command": "reservation-del",
  "arguments": {
    "subnet-id": 1,
    "ip-address": "192.0.2.202"
  }
}
```

An example deletion by (subnet-id, identifier-type, identifier) looks as follows:

```
{
  "command": "reservation-del",
  "arguments": {
    "subnet-id": 4,
    "identifier-type": "hw-address",
    "identifier": "01:02:03:04:05:06"
  }
}
```

reservation-del returns result 0 when the host deletion was successful or 1 if it was not. A descriptive text is provided in case of error. Example results look as follows:

```
{
  "result": 1,
  "text": "Host not deleted (not found)."
```

```
{
  "result": 0,
  "text": "Host deleted."
```

```
{
  "result": 1,
  "text": "Unable to delete a host because there is no hosts-database
         configured."
```

15.4.5 lease_cmds: Lease Commands

This section describes the hook library with commands used to manage leases. Kea provides a way to store lease information in several backends (memfile, MySQL, PostgreSQL and Cassandra), and this library provides a interface that can manipulate leases in a unified, safe way. In particular, it allows things previously impossible: lease manipulation in memfile while Kea is running, sanity check changes, lease existence checks, and removal of all leases belonging to a specific subnet. It can also catch more obscure errors, like an attempt to add a lease with a subnet-id that does not exist in the configuration, or configuring a lease to use an address that is outside of the subnet to which it is supposed to belong. The library also provides a non-programmatic way to manage user contexts associated with leases.

Note

This library may only be loaded by the **kea-dhcp4** or the **kea-dhcp6** process.



There are many use cases where an administrative command may be useful; for example, during migration between servers or different vendors, when a certain network is being retired, or when a device has been disconnected and the sysadmin knows for sure that it will not be coming back. The "get" queries may be useful for automating certain management and monitoring tasks. They can also act as preparatory steps for lease updates and removals.

This library provides the following commands:

- **lease4-add** - adds a new IPv4 lease;
- **lease6-add** - adds a new IPv6 lease;
- **lease6-bulk-apply** - creates, updates and/or deletes multiple IPv6 leases in a single transaction;
- **lease4-get** - checks whether an IPv4 lease with the specified parameters exists and returns it if it does;
- **lease6-get** - checks whether an IPv6 lease with the specified parameters exists and returns it if it does;
- **lease4-get-all** - returns all IPv4 leases or all IPv4 leases for the specified subnets;
- **lease6-get-all** - returns all IPv6 leases or all IPv6 leases for the specified subnets;
- **lease4-get-page** - returns a set ("page") of leases from the list of all IPv4 leases in the database. By iterating through the pages it is possible to retrieve all the leases;
- **lease6-get-page** - returns a set ("page") of leases from the list of all IPv6 leases in the database. By iterating through the pages it is possible to retrieve all the leases;
- **lease4-del** - deletes an IPv4 lease with the specified parameters;
- **lease6-del** - deletes an IPv6 lease with the specified parameters;
- **lease4-update** - updates an IPv4 lease;
- **lease6-update** - updates an IPv6 lease;
- **lease4-wipe** - removes all leases from a specific IPv4 subnet or from all subnets;
- **lease6-wipe** - removes all leases from a specific IPv6 subnet or from all subnets;

The lease commands library is part of the open source code and is available to every Kea user.

All commands use JSON syntax and can be issued either using control channel (see Chapter 17) or Control Agent (see Chapter 7).

The library can be loaded in the same way as other hook libraries, and it does not take any parameters. It supports both DHCPv4 and DHCPv6 servers.

```
"Dhcp6": {
  "hooks-libraries": [
    {
      "library": "/path/libdhcp_lease_cmds.so"
    }
    ...
  ]
}
```

15.4.5.1 lease4-add, lease6-add Commands

The **lease4-add** and **lease6-add** commands allow for the creation of a new lease. Typically Kea creates a lease when it first sees a new device; however, sometimes it may be convenient to create the lease manually. The **lease4-add** command requires at least two parameters: an IPv4 address and an identifier, i.e. hardware (MAC) address. A third parameter, subnet-id, is optional. If the subnet-id is not specified or the specified value is 0, Kea will try to determine the value by running a subnet-selection procedure. If specified, however, its value must match the existing subnet. The simplest successful call might look as follows:



```
{
  "command": "lease4-add",
  "arguments": {
    "ip-address": "192.0.2.202",
    "hw-address": "1a:1b:1c:1d:1e:1f"
  }
}
```

The **lease6-add** command requires three parameters: an IPv6 address, an IAID value (identity association identifier, a value sent by clients), and a DUID. As with lease4-add, the subnet-id parameter is optional. If the subnet-id is not specified or the provided value is 0, Kea will try to determine the value by running a subnet-selection procedure. If specified, however, its value must match the existing subnet. For example:

```
{
  "command": "lease6-add",
  "arguments": {
    "subnet-id": 66,
    "ip-address": "2001:db8::3",
    "duid": "1a:1b:1c:1d:1e:1f:20:21:22:23:24",
    "iaid": 1234
  }
}
```

lease6-add can also be used to add leases for IPv6 prefixes. In this case there are three additional parameters that must be specified: subnet-id, type (set to value of "IA_PD"), and prefix length. The actual prefix is set using ip-address field. Note that Kea cannot guess subnet-id values for prefixes; they must be specified explicitly. For example, to configure a lease for prefix 2001:db8:abcd::/48, the following command can be used:

```
{
  "command": "lease6-add",
  "arguments": {
    "subnet-id": 66,
    "type": "IA_PD",
    "ip-address": "2001:db8:abcd::",
    "prefix-len": 48,
    "duid": "1a:1b:1c:1d:1e:1f:20:21:22:23:24",
    "iaid": 1234
  }
}
```

The commands can take a number of additional optional parameters:

- **valid-lft** - specifies the lifetime of the lease, expressed in seconds. If not specified, the value configured in the subnet related to the specified subnet-id is used.
- **expire** - creates a timestamp of the lease expiration time, expressed in unix format (seconds since 1 Jan 1970). If not specified, the default value is now + the lease lifetime (the value of valid-lft).
- **fqdn-fwd** - specifies whether the lease should be marked as if a forward DNS update were conducted. Note this only affects the data stored in the lease database, and no DNS update will be performed. If configured, a DNS update to remove the A or AAAA records will be conducted when the lease is removed due to expiration or being released by a client. If not specified, the default value is false. The hostname parameter must be specified if fqdn-fwd is set to true.
- **fqdn-rev** - specifies whether the lease should be marked as if reverse DNS update were conducted. Note this only affects the data stored in the lease database, and no DNS update will be performed.. If configured, a DNS update to remove the PTR record will be conducted when the lease is removed due to expiration or being released by a client. If not specified, the default value is false. The hostname parameter must be specified if fqdn-fwd is set to true.
- **hostname** - specifies the hostname to be associated with this lease. Its value must be non-empty if either fqdn-fwd or fwn-rev are set to true. If not specified, the default value is an empty string.



- **hw-address** - optionally specifies a hardware (MAC) address for an IPv6 lease. It is a mandatory parameter for an IPv4 lease.
- **client-id** - optionally specifies a client identifier for an IPv4 lease.
- **preferred-lft** - optionally specifies a preferred lifetime for IPv6 leases. If not specified, the value configured for the subnet corresponding to the specified subnet-id is used. This parameter is not used when adding an IPv4 lease.
- **user-context** - specifies the user context to be associated with this lease. It must be a JSON map.

Here is an example of a more complex lease addition:

```
{
  "command": "lease6-add",
  "arguments": {
    "subnet-id": 66,
    "ip-address": "2001:db8::3",
    "duid": "01:02:03:04:05:06:07:08",
    "iaid": 1234,
    "hw-address": "1a:1b:1c:1d:1e:1f",
    "preferred-lft": 500,
    "valid-lft": 1000,
    "expire": 12345678,
    "fqdn-fwd": true,
    "fqdn-rev": true,
    "hostname": "urania.example.org",
    "user-context": { "version": 1 }
  }
}
```

The command returns a status that indicates either a success (result 0) or a failure (result 1). A failed command always includes a text parameter that explains the cause of failure. For example:

```
{ "result": 0, "text": "Lease added." }
```

Example failure:

```
{ "result": 1, "text": "missing parameter 'ip-address' (<string>:3:19)" }
```

15.4.5.1.1 lease6-bulk-apply

The **lease6-bulk-apply** was implemented to address the performance penalty in the High Availability when a single DHCPv6 transaction resulted in multiple lease updates sent to the partner if multiple address and/or prefix leases were allocated. Consider the case when a DHCPv6 client requests the assignment of two IPv6 addresses and two IPv6 prefixes. That may result in allocation of 4 leases. In addition, the DHCPv6 may assign different address than requested by the client during the renew or rebind and delete the leases previously used by this client. The are 6 of lease changes sent between the HA partners is in this case. Sending these updates in individual commands, e.g. **lease6-update** is highly inefficient and produces unnecessary delays in communication between the HA partners and in sending the response to the DHCPv6 client.

The **lease6-bulk-apply** command deals with this problem by aggregating all lease changes in a single command. Both deleted leases and new/updated leases are conveyed in a single command. The receiving server iterates over the deleted leases and deletes them from its lease database. Next, it iterates over the new/updated leases and adds them to the database or updates them if they already exist.

Even though the High Availability is the major application for this command, it can be freely used in all cases when it is desired to send multiple lease changes in a single command.

In the following example, we ask to delete two leases and to add or update two other leases in the database:

```
{
  "command": "lease6-bulk-apply",
  "arguments": {
```



```
"deleted-leases": [
  {
    "ip-address": "2001:db8:abcd:",
    "type": "IA_PD",
  },
  {
    "ip-address": "2001:db8:abcd::234",
    "type": "IA_NA",
  }
],
"leases": [
  {
    "subnet-id": 66,
    "ip-address": "2001:db8:cafe:",
    "type": "IA_PD",
    ...
  },
  {
    "subnet-id": 66,
    "ip-address": "2001:db8:abcd::333",
    "type": "IA_NA",
    ...
  }
]
}
```

If any of the leases is malformed, no leases changes are applied to the lease database. If the leases are well formed but there is a failure to apply any of the lease changes to the database, the command will continue to be processed for other leases. All the leases for which the command was unable to apply the changes in the database will be listed in the response.

For example:

```
{
  "result": 0,
  "text": "Bulk apply of 2 IPv6 leases completed".
  "arguments": {
    "failed-deleted-leases": [
      {
        "ip-address": "2001:db8:abcd:",
        "type": "IA_PD",
        "result": 3,
        "error-message": "no lease found"
      }
    ],
    "failed-leases": [
      {
        "ip-address": "2001:db8:cafe:",
        "type": "IA_PD",
        "result": 1,
        "error-message": "unable to communicate with the lease database"
      }
    ]
  }
}
```

The response above indicates that the hooks library was unable to delete the lease for prefix "2001:db8:abcd:" and add or update the lease for prefix "2001:db8:cafe:". However, there are two other lease changes which have been applied as indicated by the text message. The **result** is the status constant that indicates the type of the error experienced for the particular lease. The meaning of the returned codes are the same as the results returned for the commands. In particular, the result of 1 indicates an error while processing the lease, e.g. a communication error with the database. The result of 3 indicates that an attempt to delete the lease was unsuccessful because such lease doesn't exist (empty result).



15.4.5.1.2 lease4-get, lease6-get Commands

lease4-get or **lease6-get** can be used to query the lease database and retrieve existing leases. There are two types of parameters the **lease4-get** command supports: (address) or (subnet-id, identifier-type, identifier). There are also two types for **lease6-get**: (address,type) or (subnet-id, identifier-type, identifier, IAID, type). The first type of query is used when the address (either IPv4 or IPv6) is known, but the details of the lease are not; one common use case of this type of query is to find out whether a given address is being used. The second query uses identifiers; currently supported identifiers for leases are: "hw-address" (IPv4 only), "client-id" (IPv4 only), and "duid" (IPv6 only).

An example **lease4-get** command for getting a lease using an IPv4 address is:

```
{
  "command": "lease4-get",
  "arguments": {
    "ip-address": "192.0.2.1"
  }
}
```

An example of the **lease6-get** query is:

```
{
  "command": "lease6-get",
  "arguments": {
    "ip-address": "2001:db8:1234:ab::",
    "type": "IA_PD"
  }
}
```

An example query by "hw-address" for an IPv4 lease looks as follows:

```
{
  "command": "lease4-get",
  "arguments": {
    "identifier-type": "hw-address",
    "identifier": "08:08:08:08:08:08",
    "subnet-id": 44
  }
}
```

An example query by "client-id" for an IPv4 lease looks as follows:

```
{
  "command": "lease4-get",
  "arguments": {
    "identifier-type": "client-id",
    "identifier": "01:01:02:03:04:05:06",
    "subnet-id": 44
  }
}
```

An example query by (subnet-id, identifier-type, identifier, iaaid, type) for an IPv6 lease is:

```
{
  "command": "lease4-get",
  "arguments": {
    "identifier-type": "duid",
    "identifier": "08:08:08:08:08:08",
    "iaaid": 1234567,
    "type": "IA_NA",
    "subnet-id": 44
  }
}
```



The type is an optional parameter. Supported values are: IA_NA (non-temporary address) and IA_PD (IPv6 prefix). If not specified, IA_NA is assumed.

leaseX-get returns a result that indicates a result of the operation and lease details, if found. It has one of the following values: 0 (success), 1 (error), or 2 (empty). An empty result means that a query has been completed properly, but the object (a lease in this case) has not been found. The lease parameters, if found, are returned as arguments.

An example result returned when the host was found:

```
{
  "arguments": {
    "client-id": "42:42:42:42:42:42:42:42",
    "cltt": 12345678,
    "fqdn-fwd": false,
    "fqdn-rev": true,
    "hostname": "myhost.example.com.",
    "hw-address": "08:08:08:08:08:08",
    "ip-address": "192.0.2.1",
    "state": 0,
    "subnet-id": 44,
    "valid-lft": 3600
  },
  "result": 0,
  "text": "IPv4 lease found."
}
```

15.4.5.1.3 lease4-get-all, lease6-get-all Commands

lease4-get-all and **lease6-get-all** are used to retrieve all IPv4 or IPv6 leases, or all leases for the specified set of subnets. All leases are returned when there are no arguments specified with the command, as in the following example:

```
{
  "command": "lease4-get-all"
}
```

If the arguments are provided, it is expected that they contain the "subnets" parameter, which is a list of subnet identifiers for which the leases should be returned. For example, in order to retrieve all IPv6 leases belonging to the subnets with identifiers 1, 2, 3, and 4:

```
{
  "command": "lease6-get-all",
  "arguments": {
    "subnets": [ 1, 2, 3, 4 ]
  }
}
```

The returned response contains a detailed list of leases in the following format:

```
{
  "arguments": {
    "leases": [
      {
        "cltt": 12345678,
        "duid": "42:42:42:42:42:42:42:42",
        "fqdn-fwd": false,
        "fqdn-rev": true,
        "hostname": "myhost.example.com.",
        "hw-address": "08:08:08:08:08:08",
        "iaid": 1,
        "ip-address": "2001:db8:2::1",
        "preferred-lft": 500,

```



```
        "state": 0,
        "subnet-id": 44,
        "type": "IA_NA",
        "valid-lft": 3600
    },
    {
        "cltt": 12345678,
        "duid": "21:21:21:21:21:21:21:21",
        "fqdn-fwd": false,
        "fqdn-rev": true,
        "hostname": "",
        "iaid": 1,
        "ip-address": "2001:db8:0:0:2::",
        "preferred-lft": 500,
        "prefix-len": 80,
        "state": 0,
        "subnet-id": 44,
        "type": "IA_PD",
        "valid-lft": 3600
    }
]
},
"result": 0,
"text": "2 IPv6 lease(s) found."
}
```

**Warning**

The **lease4-get-all** and **lease6-get-all** commands may result in very large responses. This may have a negative impact on the DHCP server's responsiveness while the response is generated and transmitted over the control channel, as the server imposes no restriction on the number of leases returned as a result of this command.

15.4.5.1.4 lease4-get-page, lease6-get-page Commands

The **lease4-get-all** and **lease6-get-all** commands may result in very large responses; generating such a response may consume CPU bandwidth as well as memory. It may even cause the server to become unresponsive. In case of large lease databases it is usually better to retrieve leases in chunks, using the paging mechanism. **lease4-get-page** and **lease6-get-page** implement a paging mechanism for DHCPv4 and DHCPv6 servers respectively. The following command retrieves the first 1024 IPv4 leases:

```
{
  "command": "lease4-get-page",
  "arguments": {
    "from": "start",
    "limit": 1024
  }
}
```

The keyword **start** denotes that the first page of leases should be retrieved. Alternatively, an IPv4 zero address can be specified to retrieve the first page:

```
{
  "command": "lease4-get-page",
  "arguments": {
    "from": "0.0.0.0",
    "limit": 1024
  }
}
```

Similarly, the IPv6 zero address can be specified in the **lease6-get-page** command:



```
{
  "command": "lease6-get-page",
  "arguments": {
    "from": "::",
    "limit": 6
  }
}
```

The response has the following structure:

```
{
  "arguments": {
    "leases": [
      {
        "ip-address": "2001:db8:2::1",
        ...
      },
      {
        "ip-address": "2001:db8:2::9",
        ...
      },
      {
        "ip-address": "2001:db8:3::1",
        ...
      },
      {
        "ip-address": "2001:db8:5::3",
        ...
      },
      {
        "ip-address": "2001:db8:4::1",
        ...
      },
      {
        "ip-address": "2001:db8:2::7",
        ...
      }
    ],
    "count": 6
  },
  "result": 0,
  "text": "6 IPv6 lease(s) found."
}
```

Note that the leases' details were excluded from the response above for brevity.

Generally, the returned list is not sorted in any particular order. Some lease database backends may sort leases in ascending order of addresses, but the controlling client must not rely on this behavior. In cases of highly distributed databases, such as Cassandra, ordering may be inefficient or even impossible.

The **count** parameter contains the number of returned leases on the page.

To fetch the next page, the client must use the last address of the current page as an input to the next **lease4-get-page** or **lease6-get-page** command call. In this example it is:

```
{
  "command": "lease6-get-page",
  "arguments": {
    "from": "2001:db8:2::7",
    "count": 6
  }
}
```



because 2001:db8:2::7 is the last address on the current page.

The client may assume that it has reached the last page when the **count** value is lower than that specified in the command; this includes the case when the **count** is equal to 0, meaning that no leases were found.

15.4.5.1.5 lease4-del, lease6-del Commands

leaseX-del can be used to delete a lease from the lease database. There are two types of parameters this command supports, similar to leaseX-get commands: (address) for both v4 and v6, (subnet-id, identifier-type, identifier) for v4, and (subnet-id, identifier-type, identifier, type, IAID) for v6. The first type of query is used when the address (either IPv4 or IPv6) is known, but the details of the lease are not. One common use case is where an administrator wants a specified address to no longer be used. The second form of the command uses identifiers. For maximum flexibility, this interface uses identifiers as a pair of values: type and the actual identifier. The currently supported identifiers are "hw-address" (IPv4 only), "client-id" (IPv4 only), and "duid" (IPv6 only).

An example command for deleting a lease by address is:

```
{
  "command": "lease4-del",
  "arguments": {
    "ip-address": "192.0.2.202"
  }
}
```

An example IPv4 lease deletion by "hw-address" is:

```
{
  "command": "lease4-del",
  "arguments": {
    "identifier": "08:08:08:08:08:08",
    "identifier-type": "hw-address",
    "subnet-id": 44
  }
}
```

leaseX-del returns a result that indicates the outcome of the operation. It has one of the following values: 0 (success), 1 (error), or 3 (empty). The empty result means that a query has been completed properly, but the object (a lease in this case) has not been found.

15.4.5.1.6 lease4-update, lease6-update Commands

The **lease4-update** and **lease6-update** commands can be used to update existing leases. Since all lease database backends are indexed by IP addresses, it is not possible to update an address, but all other fields may be altered. If an address needs to be changed, please use **leaseX-del** followed by **leaseX-add**.

The subnet-id parameter is optional. If not specified, or if the specified value is 0, Kea will try to determine its value by running a subnet-selection procedure. If specified, however, its value must match the existing subnet.

The optional boolean parameter "force-create" specifies whether the lease should be created if it doesn't exist in the database. It defaults to false, which indicates that the lease is not created if it doesn't exist. In such a case, an error is returned as a result of trying to update a non-existing lease. If the "force-create" parameter is set to true and the updated lease does not exist, the new lease is created as a result of receiving the **leaseX-update**.

An example of a command to update an IPv4 lease is:

```
{
  "command": "lease4-update",
  "arguments": {
    "ip-address": "192.0.2.1",
    "hostname": "newhostname.example.org",
    "hw-address": "1a:1b:1c:1d:1e:1f",
  }
}
```



```
"subnet-id": 44,  
"force-create": true  
}  
}
```

An example of a command to update an IPv6 lease is:

```
{  
  "command": "lease6-update",  
  "arguments": {  
    "ip-address": "2001:db8::1",  
    "duid": "88:88:88:88:88:88:88:88",  
    "iaid": 7654321,  
    "hostname": "newhostname.example.org",  
    "subnet-id": 66,  
    "force-create": false  
  }  
}
```

15.4.5.1.7 lease4-wipe, lease6-wipe Commands

lease4-wipe and **lease6-wipe** are designed to remove all leases associated with a given subnet. This administrative task is expected to be used when an existing subnet is being retired. Note that the leases are not properly expired: no DNS updates are carried out, no log messages are created, and hooks are not called for the leases being removed.

An example of **lease4-wipe** is:

```
{  
  "command": "lease4-wipe",  
  "arguments": {  
    "subnet-id": 44  
  }  
}
```

An example of **lease6-wipe** is:

```
{  
  "command": "lease6-wipe",  
  "arguments": {  
    "subnet-id": 66  
  }  
}
```

The commands return a text description of the number of leases removed, plus the status code 0 (success) if any leases were removed or 2 (empty) if there were no leases. Status code 1 (error) may be returned if the parameters are incorrect or some other exception is encountered.

Subnet-id 0 has a special meaning; it tells Kea to delete leases from all configured subnets. Also, the subnet-id parameter may be omitted. If not specified, leases from all subnets are wiped.

Note: not all backends support this command.

15.4.6 subnet_cmds: Subnet Commands

This section describes a hook application that offers a number of new commands used to query and manipulate subnet and shared network configurations in Kea. This application is very useful in deployments with a large number of subnets being managed by the DHCP servers and when the subnets are frequently updated. The commands offer lightweight approach for manipulating subnets without a need to fully reconfigure the server and without affecting existing servers' configurations. An ability to manage shared networks (listing, retrieving details, adding new ones, removing existing ones, adding subnets to and removing from shared networks) is also provided.

Currently this library is only available to ISC customers with a support contract.

**Note**

This library may only be loaded by **kea-dhcp4** or **kea-dhcp6** process.

The following commands are currently supported:

- **subnet4-list/subnet6-list**: lists all configured subnets
- **subnet4-get/subnet6-get**: retrieves detailed information about a specified subnet
- **subnet4-add/subnet6-add**: adds new subnet into server's configuration
- **subnet4-update/subnet6-update**: updates subnet in server's configuration
- **subnet4-del/subnet6-del**: removes a subnet from the server's configuration
- **network4-list/network6-list**: lists all configured shared networks
- **network4-get/network6-get**: retrieves detailed information about specified shared network
- **network4-add/network6-add**: adds a new shared network to the server's configuration
- **network4-del/network6-del**: removes a shared network from the server's configuration
- **network4-subnet-add/network6-subnet-add**: adds existing subnet to existing shared network
- **network4-subnet-del/network6-subnet-del**: removes a subnet from existing shared network and demotes it to a plain subnet.

15.4.6.1 subnet4-list command

This command is used to list all currently configured subnets. The subnets are returned in a brief form, i.e. a subnet identifier and subnet prefix is included for each subnet. In order to retrieve the detailed information about the subnet the **subnet4-get** should be used.

This command has the simple structure:

```
{
  "command": "subnet4-list"
}
```

The list of subnets returned as a result of this command is returned in the following format:

```
{
  "result": 0,
  "text": "2 IPv4 subnets found",
  "arguments": {
    "subnets": [
      {
        "id": 10,
        "subnet": "10.0.0.0/8"
      },
      {
        "id": 100,
        "subnet": "192.0.2.0/24"
      }
    ]
  }
}
```

If no IPv4 subnets are found, an error code is returned along with the error description.



15.4.6.2 subnet6-list command

This command is used to list all currently configured subnets. The subnets are returned in a brief form, i.e. a subnet identifier and subnet prefix is included for each subnet. In order to retrieve the detailed information about the subnet the **subnet6-get** should be used.

This command has the simple structure:

```
{
  "command": "subnet6-list"
}
```

The list of subnets returned as a result of this command is returned in the following format:

```
{
  "result": 0,
  "text": "2 IPv6 subnets found",
  "arguments": {
    "subnets": [
      {
        "id": 11,
        "subnet": "2001:db8:1::/64"
      },
      {
        "id": 233,
        "subnet": "3000::/16"
      }
    ]
  }
}
```

If no IPv6 subnets are found, an error code is returned along with the error description.

15.4.6.3 subnet4-get command

This command is used to retrieve detailed information about the specified subnet. This command usually follows the **subnet4-list**, which is used to discover available subnets with their respective subnet identifiers and prefixes. Any of those parameters can be then used in **subnet4-get** to fetch subnet information:

```
{
  "command": "subnet4-get",
  "arguments": {
    "id": 10
  }
}
```

or

```
{
  "command": "subnet4-get",
  "arguments": {
    "subnet": "10.0.0.0/8"
  }
}
```

If the subnet exists the response will be similar to this:

```
{
  "result": 0,
  "text": "Info about IPv4 subnet 10.0.0.0/8 (id 10) returned",
  "arguments": {
    "subnets": [
```



```
{
  "subnet": "10.0.0.0/8",
  "id": 1,
  "option-data": [
    ....
  ]
  ...
}
]
```

15.4.6.4 subnet6-get command

This command is used to retrieve detailed information about the specified subnet. This command usually follows the **subnet6-list**, which is used to discover available subnets with their respective subnet identifiers and prefixes. Any of those parameters can be then used in **subnet6-get** to fetch subnet information:

```
{
  "command": "subnet6-get",
  "arguments": {
    "id": 11
  }
}
```

or

```
{
  "command": "subnet6-get",
  "arguments": {
    "subnet": "2001:db8:1::/64"
  }
}
```

If the subnet exists the response will be similar to this:

```
{
  "result": 0,
  "text": "Info about IPv6 subnet 2001:db8:1::/64 (id 11) returned",
  "arguments": {
    "subnets": [
      {
        "subnet": "2001:db8:1::/64",
        "id": 1,
        "option-data": [
          ...
        ]
      }
    ]
  }
}
```

15.4.6.5 subnet4-add

This command is used to create and add new subnet to the existing server configuration. This operation has no impact on other subnets. The subnet identifier must be specified and must be unique among all subnets. If the identifier or a subnet prefix is not unique an error is reported and the subnet is not added.



The subnet information within this command has the same structure as the subnet information in the server configuration file with the exception that static host reservations must not be specified within **subnet4-add**. The commands described in Section 15.4.4 should be used to add, remove and modify static reservations.

```
{
  "command": "subnet4-add",
  "arguments": {
    "subnet4": [ {
      "id": 123,
      "subnet": "10.20.30.0/24",
      ...
    } ]
  }
}
```

The response to this command has the following structure:

```
{
  "result": 0,
  "text": "IPv4 subnet added",
  "arguments": {
    "subnet4": [
      {
        "id": 123,
        "subnet": "10.20.30.0/24"
      }
    ]
  }
}
```

15.4.6.6 subnet6-add

This command is used to create and add new subnet to the existing server configuration. This operation has no impact on other subnets. The subnet identifier must be specified and must be unique among all subnets. If the identifier or a subnet prefix is not unique an error is reported and the subnet is not added.

The subnet information within this command has the same structure as the subnet information in the server configuration file with the exception that static host reservations must not be specified within **subnet6-add**. The commands described in Section 15.4.4 should be used to add, remove and modify static reservations.

```
{
  "command": "subnet6-add",
  "arguments": {
    "subnet6": [ {
      "id": 234,
      "subnet": "2001:db8:1::/64",
      ...
    } ]
  }
}
```

The response to this command has the following structure:

```
{
  "result": 0,
  "text": "IPv6 subnet added",
  "arguments": {
    "subnet6": [
      {
        "id": 234,
        "subnet": "2001:db8:1::/64"
      }
    ]
  }
}
```



```
    }
  ]
}
}
```

It is recommended, but not mandatory to specify subnet id. If not specified, Kea will try to assign the next subnet-id value. This automatic ID value generator is simple. It returns a previously automatically assigned value increased by 1. This works well, unless you manually create a subnet with a value bigger than previously used. For example, if you call `subnet4-add` five times, each without id, Kea will assign IDs: 1,2,3,4 and 5 and it will work just fine. However, if you try to call `subnet4-add` five times, with the first subnet having subnet-id of value 3 and remaining ones having no subnet-id, it will fail. The first command (with explicit value) will use subnet-id 3, the second command will create a subnet with id of 1, the third will use value of 2 and finally the fourth will have the subnet-id value auto-generated as 3. However, since there is already a subnet with that id, it will fail.

The general recommendation is to either: never use explicit values (so the auto-generated values will always work) or always use explicit values (so the auto-generation is never used). You can mix those two approaches only if you understand how the internal automatic subnet-id generation works.

Note Subnet IDs must be greater than zero and less than 4294967295.

15.4.6.7 subnet4-update

This command is used to update a subnet in the existing server configuration. This operation has no impact on other subnets. The subnet identifier is used to identify the subnet to replace, it must be specified and must be unique among all subnets. The subnet prefix should not be updated.

The subnet information within this command has the same structure as the subnet information in the server configuration file with the exception that static host reservations must not be specified within **subnet4-update**. The commands described in Section 15.4.4 should be used to update, remove and modify static reservations.

```
{
  "command": "subnet4-update",
  "arguments": {
    "subnet4": [ {
      "id": 123,
      "subnet": "10.20.30.0/24",
      ...
    } ]
  }
}
```

The response to this command has the following structure:

```
{
  "result": 0,
  "text": "IPv4 subnet updated",
  "arguments": {
    "subnet4": [
      {
        "id": 123,
        "subnet": "10.20.30.0/24"
      }
    ]
  }
}
```



15.4.6.8 subnet6-update

This command is used to update a subnet in the existing server configuration. This operation has no impact on other subnets. The subnet identifier is used to identify the subnet to replace, it must be specified and must be unique among all subnets. The subnet prefix should not be updated.

The subnet information within this command has the same structure as the subnet information in the server configuration file with the exception that static host reservations must not be specified within **subnet6-update**. The commands described in Section 15.4.4 should be used to update, remove and modify static reservations.

```
{
  "command": "subnet6-update",
  "arguments": {
    "subnet6": [ {
      "id": 234,
      "subnet": "2001:db8:1::/64",
      ...
    } ]
  }
}
```

The response to this command has the following structure:

```
{
  "result": 0,
  "text": "IPv6 subnet updated",
  "arguments": {
    "subnet6": [
      {
        "id": 234,
        "subnet": "2001:db8:1::/64"
      }
    ]
  }
}
```

15.4.6.9 subnet4-del command

This command is used to remove a subnet from the server's configuration. This command has no effect on other configured subnets but removing a subnet has certain implications which the server's administrator should be aware of.

In most cases the server has assigned some leases to the clients belonging to the subnet. The server may also be configured with static host reservations which are associated with this subnet. The current implementation of the **subnet4-del** removes neither the leases nor host reservations associated with a subnet. This is the safest approach because the server doesn't lose track of leases assigned to the clients from this subnet. However, removal of the subnet may still cause configuration errors and conflicts. For example: after removal of the subnet, the server administrator may update a new subnet with the ID used previously for the removed subnet. This means that the existing leases and static reservations will be in conflict with this new subnet. Thus, we recommend that this command is used with extreme caution.

This command can also be used to completely delete an IPv4 subnet that is part of a shared network. If you want to simply remove the subnet from a shared network and keep the subnet configuration, use **network4-subnet-del** command instead.

The command has the following structure:

```
{
  "command": "subnet4-del",
  "arguments": {
    "id": 123
  }
}
```



The example successful response may look like this:

```
{
  "result": 0,
  "text": "IPv4 subnet 192.0.2.0/24 (id 123) deleted",
  "arguments": {
    "subnets": [
      {
        "id": 123,
        "subnet": "192.0.2.0/24"
      }
    ]
  }
}
```

15.4.6.10 subnet6-del command

This command is used to remove a subnet from the server's configuration. This command has no effect on other configured subnets but removing a subnet has certain implications which the server's administrator should be aware of.

In most cases the server has assigned some leases to the clients belonging to the subnet. The server may also be configured with static host reservations which are associated with this subnet. The current implementation of the **subnet6-del** removes neither the leases nor host reservations associated with a subnet. This is the safest approach because the server doesn't lose track of leases assigned to the clients from this subnet. However, removal of the subnet may still cause configuration errors and conflicts. For example: after removal of the subnet, the server administrator may add a new subnet with the ID used previously for the removed subnet. This means that the existing leases and static reservations will be in conflict with this new subnet. Thus, we recommend that this command is used with extreme caution.

This command can also be used to completely delete an IPv6 subnet that is part of a shared network. If you want to simply remove the subnet from a shared network and keep the subnet configuration, use **network6-subnet-del** command instead.

The command has the following structure:

```
{
  "command": "subnet6-del",
  "arguments": {
    "id": 234
  }
}
```

The example successful response may look like this:

```
{
  "result": 0,
  "text": "IPv6 subnet 2001:db8:1::/64 (id 234) deleted",
  "subnets": [
    {
      "id": 234,
      "subnet": "2001:db8:1::/64"
    }
  ]
}
```

15.4.6.11 network4-list, network6-list commands

These commands are used to retrieve full list of currently configured shared networks. The list contains only very basic information about each shared network. If more details are needed, please use **network4-get** or **network6-get** to retrieve all information available. This command does not require any parameters and its invocation is very simple:



```
{
  "command": "network4-list"
}
```

An example response for **network4-list** looks as follows:

```
{
  "arguments": {
    "shared-networks": [
      { "name": "floor1" },
      { "name": "office" }
    ]
  },
  "result": 0,
  "text": "2 IPv4 network(s) found"
}
```

network6-list follows exactly the same syntax for both the query and the response.

15.4.6.12 network4-get, network6-get commands

These commands are used to retrieve detailed information about shared networks, including subnets currently being part of a given network. Both commands take one mandatory parameter **name**, which specify the name of shared network. An example command to retrieve details about IPv4 shared network with a name "floor13" looks as follows:

```
{
  "command": "network4-get",
  "arguments": {
    "name": "floor13"
  }
}
```

An example response could look as follows:

```
{
  "result": 0,
  "text": "Info about IPv4 shared network 'floor13' returned",
  "arguments": {
    "shared-networks": [
      {
        "match-client-id": true,
        "name": "floor13",
        "option-data": [ ],
        "rebind-timer": 90,
        "relay": {
          "ip-address": "0.0.0.0"
        },
        "renew-timer": 60,
        "reservation-mode": "all",
        "subnet4": [
          {
            "subnet": "192.0.2.0/24",
            "id": 5,
            // many other subnet specific details here
          },
          {
            "id": 6,
            "subnet": "192.0.3.0/31",
            // many other subnet specific details here
          }
        ]
      }
    ]
  }
}
```



```
    ],
    "valid-lifetime": 120
  }
]
}
```

Note that actual response contains many additional fields that are omitted here for clarity. The response format is exactly the same as used in **config-get**, just is limited to returning shared networks information.

15.4.6.13 network4-add, network6-add commands

These commands are used to add a new shared network. New network has to have unique name. This command requires one parameter **shared-networks**, which is a list and should contain exactly one entry that defines the network. The only mandatory element for a network is its name. Although it does not make operational sense, it is allowed to add an empty shared network that does not have any subnets in it. That is allowed for testing purposes, but having empty networks (or with only one subnet) is discouraged in production environments. For details regarding syntax, see Section 8.4 and Section 9.4.

Note

As opposed to parameter inheritance during full new configuration processing, this command does not fully handle parameter inheritance and any missing parameters will be filled with default values, rather than inherited from global scope.

An example that showcases how to add a new IPv4 shared network looks as follows:

```
{
  "command": "network4-add",
  "arguments": {
    "shared-networks": [ {
      "name": "floor13",
      "subnet4": [
        {
          "id": 100,
          "pools": [ { "pool": "192.0.2.2-192.0.2.99" } ],
          "subnet": "192.0.2.0/24",
          "option-data": [
            {
              "name": "routers",
              "data": "192.0.2.1"
            }
          ]
        }
      ],
    },
    {
      "id": 101,
      "pools": [ { "pool": "192.0.3.2-192.0.3.99" } ],
      "subnet": "192.0.3.0/24",
      "option-data": [
        {
          "name": "routers",
          "data": "192.0.3.1"
        }
      ]
    }
  ]
} ]
}
```

Assuming there was no shared network with a name floor13 and no subnets with id 100 and 101 previously configured, the command will be successful and will return the following response:



```
{
  "arguments": {
    "shared-networks": [ { "name": "floor13" } ]
  },
  "result": 0,
  "text": "A new IPv4 shared network 'floor13' added"
}
```

The **network6-add** uses the same syntax for both the query and the response. However, there are some parameters that are IPv4-only (e.g. match-client-id) and some are IPv6-only (e.g. interface-id). The same applies to subnets within the network.

15.4.6.14 network4-del, network6-del commands

These commands are used to delete existing shared networks. Both commands take exactly one parameter 'name' that specifies the name of the network to be removed. An example invocation of **network4-del** command looks as follows:

```
{
  "command": "network4-del",
  "arguments": {
    "name": "floor13"
  }
}
```

Assuming there was such a network configured, the response will look similar to the following:

```
{
  "arguments": {
    "shared-networks": [
      {
        "name": "floor13"
      }
    ]
  },
  "result": 0,
  "text": "IPv4 shared network 'floor13' deleted"
}
```

The **network6-del** command uses exactly the same syntax for both the command and the response.

If there are any subnets belonging to the shared network being deleted, they will be demoted to a plain subnet. There is an optional parameter called **subnets-action** that, if specified, takes one of two possible values: **keep** (which is the default) and **delete**. It controls whether the subnets be demoted to plain subnets or removed. An example usage in network6-del command that deletes the shared network and all subnets in it could look like as follows:

```
{
  "command": "network4-del",
  "arguments": {
    "name": "floor13",
    "subnets-action": "delete"
  }
}
```

Alternatively, if you want to completely remove the subnets, you may use **subnet4-del** or **subnet6-del** commands.

15.4.6.15 network4-subnet-add, network6-subnet-add commands

These commands are used to add existing subnets to existing shared networks. There are several ways to add new shared network. System administrator can add the whole shared network at once, either by editing a configuration file or by calling **network4-add** or **network6-add** commands with desired subnets in it. This approach works better for completely new shared subnets.



However, there may be cases when an existing subnet is running out of addresses and needs to be extended with additional address space. In other words another subnet has to be added on top of it. For this scenario, a system administrator can use **network4-add** or **network6-add** and then add existing subnet to this newly created shared network using **network4-subnet-add** or **network6-subnet-add**.

The **network4-subnet-add** and **network6-subnet-add** commands take two parameters: **id**, which is an integer and specifies subnet-id of existing subnet to be added to a shared network; and **name**, which specifies name of the shared network the subnet will be added to. The subnet must not belong to any existing network. In case you want to reassign a subnet from one shared network to another, please use **network4-subnet-del** or **network6-subnet-del** commands first.

An example invocation of **network4-subnet-add** command looks as follows:

```
{
  "command": "network4-subnet-add",
  "arguments": {
    "name": "floor13",
    "id": 5
  }
}
```

Assuming there is a network named 'floor13', there is a subnet with subnet-id 5 and it is not a part of existing network, the command will return a response similar to the following:

```
{
  "result": 0,
  "text": "IPv4 subnet 10.0.0.0/8 (id 5) is now part of shared network 'floor1'"
}
```

The **network6-subnet-add** command uses exactly the same syntax for both the command and the response.

Note

As opposed to parameter inheritance during full new configuration processing or when adding a new shared network with new subnets, this command does not fully handle parameter inheritance and any missing parameters will be filled with default values, rather than inherited from global scope or from the shared network.

15.4.6.16 network4-subnet-del, network6-subnet-del commands

These commands are used to remove a subnet that is part of existing shared network and demote it to a plain, stand-alone subnet. If you want to remove a subnet completely, use **subnet4-del** or **subnet6-del** commands instead. The **network4-subnet-del** and **network6-subnet-del** commands take two parameters: **id**, which is an integer and specifies subnet-id of existing subnet to be removed from a shared network; and **name**, which specifies name of the shared network the subnet will be removed from.

An example invocation of the **network4-subnet-del** command looks as follows:

```
{
  "command": "network4-subnet-del",
  "arguments": {
    "name": "floor13",
    "id": 5
  }
}
```

Assuming there was a subnet with subnet-id equal to 5 that was part of a shared network named 'floor13', the response would look similar to the following:

```
{
  "result": 0,
  "text": "IPv4 subnet 10.0.0.0/8 (id 5) is now removed from shared network 'floor13'"
}
```

The **network6-subnet-del** command uses exactly the same syntax for both the command and the response.



15.4.7 class_cmds: Class Commands

This section describes the Class Commands hooks library, which exposes several control commands for manipulating client classes (part of the Kea DHCP servers' configurations) without the need to restart those servers. Using these commands it is possible to add, update, delete, and list client classes configured for a given server.

Note

This library may only be loaded by the **kea-dhcp4** or the **kea-dhcp6** process.

The Class Commands hooks library is available to premium Kea customers only.

15.4.7.1 class-add Command

The **class-add** command adds a new client class to the DHCP server configuration. This class is appended at the end of the list of classes used by the server and may depend on any of the already configured client classes.

The following example demonstrates how to add a new client class to the DHCPv4 server configuration:

```
{
  "command": "class-add",
  "arguments": {
    "client-classes": [
      {
        "name": "ipxe_efi_x64",
        "test": "option[93].hex == 0x0009",
        "next-server": "192.0.2.254",
        "server-hostname": "hal9000",
        "boot-file-name": "/dev/null"
      }
    ]
  }
}
```

Note that the **client-classes** parameter is a JSON list, but it allows only a single client class to be present.

Here is the response to the **class-add** command in our example:

```
{
  "result": 0,
  "text": "Class 'ipxe_efi_x64' added."
}
```

15.4.7.2 class-update Command

The **class-update** command updates an existing client class in the DHCP server configuration. If the client class with the given name doesn't exist, the server returns the result code of 3, which means that the server configuration is not modified and the client class does not exist. The **class-add** command must be used instead to create the new client class.

The **class-update** command has the same argument structure as the **class-add** command:

```
{
  "command": "class-update",
  "arguments": {
    "client-classes": [
      {
        "name": "ipxe_efi_x64",
        "test": "option[93].hex == 0x0017",
        "next-server": "0.0.0.0",

```



```
        "server-hostname": "xfce",
        "boot-file-name": "/dev/null"
    }
]
}
```

Here is the response for our example:

```
{
  "result": 0,
  "text": "Class 'ipxe_efi_x64' updated."
}
```

Any parameter of the client class can be modified with this command, except **name**. There is currently no way to rename the class, because the class name is used as a key for searching the class to be updated. To achieve a similar effect to renaming the class, an existing class can be removed with the **class-del** command and then added again with a different name using **class-add**. Note, however, that the class with the new name will be added at the end of the list of configured classes.

15.4.7.3 class-del Command

The **class-del** is used to remove a particular class from the server configuration. The class to be removed is identified by name. The class is not removed if there are other classes depending on it; to remove such a class, the dependent classes must be removed first.

The following is a sample command removing the **ipxe_efi_x64** class:

```
{
  "command": "class-del",
  "arguments": {
    {
      "name": "ipxe_efi_x64"
    }
  }
}
```

Here is the response to the **class-del** command in our example, when the specified client class has been found:

```
{
  "result": 0,
  "text": "Class 'ipxe_efi_x64' deleted."
}
```

If the class doesn't exist, the result of 3 is returned.

15.4.7.4 class-list Command

class-list is used to retrieve a list of all client classes. This command includes no arguments:

```
{
  "command": "class-list"
}
```

Here is the response of the server in our example, including the list of client classes:

```
{
  "result": 0,
  "text": "2 classes found",
  "arguments": {
    "client-classes": [
```



```
{
  {
    "name": "ipxe_efi_x64"
  },
  {
    "name": "pxeclient"
  }
]
}
```

Note that the returned list does not contain full class definitions, but merely class names. To retrieve full class information, the **class-get** command should be used.

15.4.7.5 class-get Command

class-get is used to retrieve detailed information about a specified class. The command structure is very simple:

```
{
  "command": "class-get",
  "arguments": {
    "name": "pxeclient"
  }
}
```

If the class with the specified name does not exist, the status code of 3 is returned. If the specified client class exists, the class details are returned in the following format:

```
{
  "result": 0,
  "text": "Class 'pxeclient' definition returned",
  "arguments": {
    "client-classes": [
      {
        "name": "pxeclient",
        "only-if-required": true,
        "test": "option[vendor-class-identifier].text == 'PXECClient'",
        "option-def": [
          {
            "name": "configfile",
            "code": 209,
            "type": "string"
          }
        ],
        "option-data": [ ],
        "next-server": "0.0.0.0",
        "server-hostname": "xfce",
        "boot-file-name": "/dev/null"
      }
    ]
  }
}
```

Note that the example above is DHCPv4-specific; the last three parameters are only returned by the DHCPv4 server and are never returned by the DHCPv6 server. Also, some of the parameters provided in this example may not be returned if they are not specified for the class. Specifically, **only-if-required**, **test**, and **option-def** are not returned if they are not specified for the class.

15.4.8 cb_cmds: Configuration Backend Commands

This section describes the **cb_cmds** hooks library which is used to manage Kea servers' configurations in the Configuration Backends. This library must be used in conjunction with the available CB hooks libraries implementing the common APIs to



create, read, update and delete (CRUD) the configuration information in the respective databases. For example: the **mysql_cb** hooks library, released in Kea 1.6.0, implements this API for MySQL. In order to manage the configuration information in the MySQL database both **mysql_cb** and **cb_cmds** libraries must be loaded by the server used for the configuration management.

The **cb_cmds** library is only available to ISC customers with a support contract.

Note

This library may only be loaded by the **kea-dhcp4** or the **kea-dhcp6** process.

15.4.8.1 Commands Structure

There are 5 types of commands supported by this library:

- **del** - delete selected object from the database, e.g. **remote-global-parameter4-del**
- **get** - fetch selected object from the database, e.g. **remote-subnet4-get**,
- **get-all** - fetch all objects of the particular type from the database, e.g. **remote-option-def4-get-all**,
- **list** - list all objects of the particular type in the database, e.g. **remote-network4-list**; this class of commands returns brief information about each object comparing to the output of **get-all**,
- **set** - creates or replaces an object of the given type in the database, e.g. **remote-option4-global-set** .

All types of commands accept optional **remote** map which selects the database instance to which the command refers. For example:

```
{
  "command": "remote-subnet4-list",
  "arguments": {
    "remote": {
      "type": "mysql",
      "host": "192.0.2.33",
      "port": 3302
    }
  }
}
```

selects the MySQL database, running on host 192.0.2.33 and port 3302, to fetch the list of subnets from. All parameters in the **remote** are optional. The **port** parameter can be only specified in conjunction with the **host**. If no parameters in the **remote** parameter are to be specified, the parameter should be omitted. In this case, the server will use the first backend listed in the **config-control** map within the configuration of the server receiving the command.

Note

In the Kea 1.6.0 release, it is possible to configure the Kea server to use only one configuration backend. Strictly speaking, it is possible to point Kea server to at most one MySQL database using the **config-control** parameter. That's why, in this release, the **remote** parameter may be omitted in the commands because the **cb_cmds** hooks library will use by default the sole backend.

15.4.8.2 Control Commands for DHCP Servers

This section describes and gives some examples of the control commands implemented by the **cb_cmds** hooks library, to manage the configuration information of the DHCPv4 and DHCPv6 servers. Many of the commands are almost identical between DHCPv4 and DHCPv6, i.e. only differ by the command name. Other commands slightly differ by the structure of the inserted data, e.g. the structure of the IPv4 subnet information is different than the structure of the IPv6 subnet. Nevertheless, they still share the structure of the command arguments and thus it makes sense to describe them together.



In the following sections, various commands are described and the usage examples are provided. In the sections jointly describing the DHCPv4 and DHCPv6 variants of the particular command we sometimes use the following notation - the **remote-subnet[46]-set** is the wildcard name for the two commands: **remote-subnet4-set** and **remote-subnet6-set**.

In addition, whenever the text in the subsequent sections refers to a DHCP command or DHCP parameter, it refers to both DHCPv4 and DHCPv6 variants. The text specific to the particular server type refers to them as: DHCPv4 command, DHCPv4 parameter, DHCPv6 command, DHCPv6 parameter etc.

15.4.8.2.1 Metadata

The typical response to the **get** or **list** command includes a list of returned objects (e.g. subnets) and each such object contains the **metadata** map with some database specific information describing this object. In other words, the metadata contains any information about the fetched object which may be useful for the administrator, but is not the part of the object specification from the DHCP server standpoint. In the Kea 1.6.0 release, the metadata is limited to the **server-tags**, which describes the associations of the object with the selected servers or all servers.

The following is the example response to the **remote-network4-list** command, which includes the metadata:

```
{
  "result": 0,
  "text": "1 IPv4 shared network(s) found.",
  "arguments": {
    "shared-networks": [
      {
        "name": "level3",
        "metadata": {
          "server-tags": [ "all" ]
        }
      }
    ],
    "count": 1
  }
}
```

Client implementations must not assume that the metadata contains only the **server-tags** parameter. In the future releases this map will be extended with additional information, e.g. object modification time, log message created during the last modification etc.

15.4.8.2.2 remote-server4-del, remote-server6-del commands

This command is used to delete the information about a selected DHCP server from the configuration database. The server is identified by a unique case insensitive server tag. For example:

```
{
  "command": "remote-server4-del",
  "arguments": {
    "servers": [
      {
        "server-tag": "server1"
      }
    ],
    "remote": {
      "type": "mysql"
    }
  }
}
```

As a result of this command, the user defined server called **server1** is removed from the database. All associations of the configuration information with this server are automatically removed from the database. The non-shareable configuration information, such as: global parameters, option definitions and global options associated with the server are removed from the database. The



shareable configuration information, i.e. the configuration elements which may be associated with more than one server, is preserved. In particular, the subnets and shared networks associated with the deleted servers are preserved. If any of the shareable configuration elements was associated only with the deleted server, this object becomes unassigned (orphaned). For example: if a subnet has been created and associated with the **server1** using the **remote-subnet4-set** command and the server1 is subsequently deleted, the subnet remains in the database but none of the servers can use this subnet. The subnet can be updated using the **remote-subnet4-set** and associated with some other server or with all servers using the special server tag "all". Such subnet can be also deleted from the database using the **remote-subnet4-del-by-id** or **remote-subnet4-del-by-prefix**, if it is no longer needed.

The following is the successful response to the **remote-server4-del** command:

```
{
  "result": 0,
  "text": "1 DHCPv4 server(s) deleted."
  "arguments": {
    "count": 1
  }
}
```

Note

The **remote-server4-del** and **remote-server6-del** commands must be used with care, because an accidental deletion of the server causes some parts of the existing configurations to be lost permanently from the database. This operation is not reversible. Re-creation of the accidentally deleted server does not revert the lost configuration for that server and such configuration must be re-created manually by the user.

15.4.8.2.3 remote-server4-get, remote-server6-get commands

This command is used to fetch the information about the selected DHCP server from the configuration database. For example:

```
{
  "command": "remote-server6-get"
  "arguments": {
    "servers": [
      {
        "server-tag": "server1"
      }
    ],
    "remote": {
      "type": "mysql"
    }
  }
}
```

This command fetches the information about the DHCPv6 server identified by the server tag **server1**. The server tag is case insensitive. A successful response returns basic information about the server, such as server tag and the user's description of the server:

```
{
  "result": 0,
  "text": "DHCP server server1 found.",
  "arguments": {
    "servers": [
      {
        "server-tag": "server1",
        "description": "A DHCPv6 server located on the first floor."
      }
    ],
    "count": 1
  }
}
```



```
}  
}
```

15.4.8.2.4 remote-server4-get-all, remote-server6-get-all commands

This command is used to fetch all user defined DHCPv4 or DHCPv6 servers from the database. The command structure is very simple:

```
{  
  "command": "remote-server4-get-all"  
  "arguments": {  
    "remote": {  
      "type": "mysql"  
    }  
  }  
}
```

The response includes basic information about each server, such as its server tag and description:

```
{  
  "result": 0,  
  "text": "DHCPv4 servers found.",  
  "arguments": {  
    "servers": [  
      {  
        "server-tag": "server1",  
        "description": "A DHCP server located on the first floor."  
      },  
      {  
        "server-tag": "server2",  
        "description": "An old DHCP server to be soon replaced."  
      }  
    ],  
    "count": 2  
  }  
}
```

15.4.8.2.5 remote-server4-set, remote-server6-set commands

This command is used to create or replace an information about a DHCP server in the database. The information about the server must be created when there is a need to differentiate the configurations used by various Kea instances connecting to the same database. Various configuration elements, e.g. global parameters, subnets etc. may be explicitly associated with the selected servers (using server tags as identifiers), allowing only these servers to use the respective configuration elements. Using the particular server tag to make such associations is only possible when the server information has been stored in the database via the **remote-server4-set** or **remote-server6-set** commands. The following command creates a new (or updates an existing) DHCPv6 server in the database:

```
{  
  "command": "remote-server6-set"  
  "arguments": {  
    "servers": [  
      {  
        "server-tag": "server1",  
        "description": "A DHCP server on the ground floor."  
      }  
    ],  
    "remote": {  
      "type": "mysql"  
    }  
  }  
}
```



```
}  
}
```

The server tag must be unique across all servers in the database. When the server information under the given server tag already exists, it is replaced with the new information. The specified server tag is case insensitive. The maximum length of the server tag is 256 characters. The following keywords are reserved and must not be used as server tags: "all" and "any".

The following is the example response to the above command:

```
{  
  "result": 0,  
  "text": "DHCPv6 server successfully set.",  
  "arguments": {  
    "servers": [  
      {  
        "server-tag": "server1",  
        "description": "A DHCP server on the ground floor."  
      }  
    ]  
  }  
}
```

15.4.8.2.6 remote-global-parameter4-del, remote-global-parameter6-del commands

This command is used to delete a global DHCP parameter from the configuration database. When the parameter is deleted from the database the server will use the value specified in the configuration file for this parameter or a default value if the parameter is not specified in the configuration file.

The following command attempts to delete the DHCPv4 **renew-timer** parameter common for all servers from the database:

```
{  
  "command": "remote-global-parameter4-del",  
  "arguments": {  
    "parameters": [ "renew-timer" ],  
    "remote": {  
      "type": "mysql"  
    },  
    "server-tags": [ "all" ]  
  }  
}
```

If the server specific parameter is to be deleted, the **server-tags** list must contain the tag of the appropriate server. There must be exactly one server tag specified in this list.

15.4.8.2.7 remote-global-parameter4-get, remote-global-parameter6-get commands

These commands are used to fetch a scalar global DHCP parameter from the configuration database.

The following command attempts to fetch the **boot-file-name** parameter for the "server1":

```
{  
  "command": "remote-global-parameter4-get",  
  "arguments": {  
    "parameters": [ "boot-file-name" ],  
    "remote": {  
      "type": "mysql"  
    },  
    "server-tags": [ "server1" ]  
  }  
}
```



The returned value has one of the four scalar types: string, integer, real or boolean. Non scalar global configuration parameters, such as map or list, are not returned by this command.

In case of the example above the string value is returned, e.g.:

```
{
  "result": 0,
  "text": "1 DHCPv4 global parameter found.",
  "arguments": {
    "parameters": {
      "boot-file-name": "/dev/null",
      "metadata": {
        "server-tags": [ "all" ]
      }
    },
    "count": 1
  }
}
```

Note that the response above indicates that the returned parameter is associated with "all" servers rather than "server1" used in the command. This indicates that there is no server1 specific value in the database. Therefore, the value shared by all servers is returned. If there was the server1 specific value in the database this value would be returned instead.

The example response for the integer value is:

```
{
  "result": 0,
  "text": "1 DHCPv4 global parameter found.",
  "arguments": {
    "parameters": {
      "renew-timer": 2000,
      "metadata": {
        "server-tags": [ "server1" ]
      }
    },
    "count": 1
  }
}
```

The real value:

```
{
  "result": 0,
  "text": "1 DHCPv4 global parameter found.",
  "arguments": {
    "parameters": {
      "t1-percent": 0.85,
      "metadata": {
        "server-tags": [ "all" ]
      }
    },
    "count": 1
  }
}
```

Finally, the boolean value:

```
{
  "result": 0,
  "text": "1 DHCPv4 global parameter found.",
  "arguments": {
    "parameters": {
      "match-client-id": true,

```



```
    "metadata": {
      "server-tags": [ "server2" ]
    }
  },
  "count": 1
}
```

15.4.8.2.8 remote-global-parameter4-get-all, remote-global-parameter6-get-all commands

These commands are used to fetch all global DHCP parameters from the database for the specified server. The following example demonstrates how to fetch all global parameters to be used by the server "server1":

```
{
  "command": "remote-global-parameter4-get-all",
  "arguments": {
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "server1" ]
  }
}
```

The example response may look as follows:

```
{
  "result": 0,
  "text": "DHCPv4 global parameters found.",
  "arguments": {
    "parameters": [
      {
        "boot-file-name": "/dev/null",
        "metadata": {
          "server-tags": [ "server1" ]
        }
      },
      {
        "match-client-id": true,
        "metadata": {
          "server-tags": [ "all" ]
        }
      }
    ],
    "count": 2
  }
}
```

The example response contains two parameters, one string parameter and one boolean parameter. The metadata returned for each parameter indicates if this parameter is specific to the "server1" or all servers. Since the **match-client-id** value is associated with "all" servers it indicates that there is no server1 specific setting for this parameter. Each parameter always has exactly one server tag associated with it, because the global parameters are non-shareable configuration elements.

Note

If the server tag is set to "all" in the command, the response will contain only the global parameters associated with the logical server "all". When the server tag points to the specific server (as in the example above), the returned list combines parameters associated with this server and all servers, but the former take precedence.



15.4.8.2.9 remote-global-parameter4-set

This command is used to create scalar global DHCP parameters in the database. If any of the parameters already exists, its value is replaced as a result of this command. It is possible to set multiple parameters within a single command, each having one of the four types: a string, integer, real or boolean. For example:

```
{
  "command": "remote-global-parameter4-set"
  "arguments": {
    "parameters": {
      "boot-file-name": "/dev/null",
      "renew-timer": 2000,
      "t1-percent": 0.85,
      "match-client-id": true
    },
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "server1" ]
  }
}
```

An error is returned if any of the parameters is not supported by the DHCP server or its type does not match. Care should be taken when multiple parameters are specified in a single command because it is possible that only some of the parameters are stored successfully and some fail. If an error occurred when processing this command, it is recommended to use **remote-global-parameter[46]-get-all** to check which of the parameters have been stored/updated successfully and which failed.

The **server-tags** list is mandatory and it must contain a single server tag or the keyword "all". In the example above, all specified parameters are associated with the "server1" server.

15.4.8.2.10 remote-network4-del, remote-network6-del commands

These commands are used to delete an IPv4 or IPv6 shared network from the database. The optional parameter **subnets-action** determines whether the subnets belonging to the deleted shared network should also be deleted or preserved. The **subnets-action** defaults to **keep**, which preserves the subnets. If it is set to **delete**, the subnets are deleted along with the shared network.

The following command:

```
{
  "command": "remote-network6-del",
  "arguments": {
    "shared-networks": [
      {
        "name": "level3"
      }
    ],
    "subnets-action": "keep",
    "remote": {
      "type": "mysql"
    }
  }
}
```

deletes the "level3" IPv6 shared network. The subnets are preserved but they are disassociated from the deleted shared network and become global. This behavior corresponds to the behavior of the **network[46]-del** commands with respect to the **subnets-action**.

Note that the **server-tags** parameter must not be used for this command.



15.4.8.2.11 remote-network4-get, remote-network6-get commands

These commands are used to retrieve the information about an IPv4 or IPv6 shared network. The optional parameter **subnets-include** denotes whether the subnets belonging to the shared network should also be returned. This parameter defaults to **no** in which case the subnets are not returned. If this parameter is set to **full**, the subnets are returned together with the shared network.

The following command fetches the "level3" IPv6 shared network along with the full information about the subnets belonging to it:

```
{
  "command": "remote-network6-get",
  "arguments": {
    "shared-networks": [
      {
        "name": "level3"
      }
    ],
    "subnets-include": "full",
    "remote": {
      "type": "mysql"
    }
  }
}
```

Note that the **server-tags** parameter must not be used for this command.

15.4.8.2.12 remote-network4-list, remote-network6-list commands

These commands are used to list all IPv4 or IPv6 shared networks for a server.

The following command retrieves all shared networks to be used by the "server1" and "server2":

```
{
  "command": "remote-network4-list"
  "arguments": {
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "server1", "server2" ]
  }
}
```

The **server-tags** parameter is mandatory and it contains one or more server tags. It may contain the keyword "all" to fetch the shared networks associated with all servers. When the **server-tags** list contains the **null** value the returned response contains a list of unassigned shared networks, i.e. the networks which are associated with no servers. For example:

```
{
  "command": "remote-network4-list"
  "arguments": {
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ null ]
  }
}
```

The example response to this command when non-null server tags are specified looks similar to this:

```
{
  "result": 0,
  "text": "3 IPv4 shared network(s) found.",
  "arguments": {
```



```
"shared-networks": [
  {
    "name": "ground floor",
    "metadata": {
      "server-tags": [ "all" ]
    }
  },
  {
    "name": "floor2",
    "metadata": {
      "server-tags": [ "server1" ]
    }
  },
  {
    "name": "floor3",
    "metadata": {
      "server-tags": [ "server2" ]
    }
  }
],
"count": 3
}
```

The returned information about each shared network merely contains the shared network name and the metadata. In order to fetch the detailed information about the selected shared network, use the **remote-network[46]-get** command.

The example response above contains three shared networks. One of the shared networks is associated with all servers, so it is included in the list of shared networks to be used by the "server1" and "server2". The remaining two shared networks are returned because one of them is associated with the "server1" and another one is associated with the "server2".

When listing unassigned shared networks, the response will look similar to this:

```
{
  "result": 0,
  "text": "1 IPv4 shared network(s) found.",
  "arguments": {
    "shared-networks": [
      {
        "name": "fancy",
        "metadata": {
          "server-tags": [ null ]
        }
      }
    ],
    "count": 1
  }
}
```

The **null** value in the metadata indicates that the returned shared network is unassigned.

15.4.8.2.13 remote-network4-set

These commands create new or replace an existing IPv4 or IPv6 shared network in the database. The structure of the shared network information is the same as in the Kea configuration file (see Section 8.4 and Section 9.4 for details), except that it is not allowed to specify subnets along with the shared network information. Including the **subnet4** or **subnet6** parameter within the shared network information will result in an error.

These commands are intended to be used for managing the shared network specific information and DHCP options. In order to associate and disassociate the subnets with the shared networks the **remote-subnet[46]-set** commands should be used.

The following command adds the IPv6 shared network "level3" to the database:



```
{
  "command": "remote-network6-set",
  "arguments": {
    "shared-networks": [
      {
        "name": "level3",
        "interface": "eth0",
        "option-data": [ {
          "name": "sntp-servers",
          "data": "2001:db8:1::1"
        } ],
      }
    ],
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "all" ]
  }
}
```

This command includes the **interface** parameter which sets the shared network level interface name. Remaining shared network level parameters, which are not specified with the command, will be marked as "unspecified" in the database. The DHCP server will use the global values for unspecified parameters or, if the global values are not specified, the default values will be used.

The **server-tags** list is mandatory for this command and it must include one or more server tags. As a result the shared network is associated with all listed servers. The shared network may be associated with all servers connecting to the database when the keyword "all" is included.

Note

Same as for other "set" commands, this command replaces the entire information about the given shared network in the database if the shared network already exists. Therefore, when sending this command, make sure to always include all parameters that must be specified for the updated shared network instance. Any unspecified parameter will be marked unspecified in the database, even if its value was present prior to sending the command.

15.4.8.2.14 remote-option-def4-del, remote-option-def6-del commands

These commands are used to delete a DHCP option definition from the database. The option definition is identified by an option code and option space. For example:

```
{
  "command": "remote-option-def6-del",
  "arguments": {
    "option-defs": [
      {
        "code": 1,
        "space": "isc"
      }
    ],
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "server1" ]
  }
}
```

deletes the definition of the option associated with the "server1", having the code of 1 and belonging to the option space "isc". The default option spaces are "dhcp4" and "dhcp6" for the DHCPv4 and DHCPv6 top level options respectively. If there is no such option explicitly associated with the server1, no option is deleted. In order to delete an option belonging to "all" servers, the keyword "all" must be used as server tag. The **server-tags** list must contain exactly one tag. It must not include the **null** value.



15.4.8.2.15 remote-option-def4-get, remote-option-def6-get commands

These commands are used to fetch a specified DHCP option definition from the database. The option definition is identified by the option code and option space. The default option spaces are "dhcp4" and "dhcp6" for the DHCPv4 and DHCPv6 top level options respectively.

The following command retrieves a DHCPv4 option definition associated with all servers, having the code of 1 and belonging to the option space "isc":

```
{
  "command": "remote-option-def4-get"
  "arguments": {
    "option-defs": [
      {
        "code": 1,
        "space": "isc"
      }
    ],
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "all" ]
  }
}
```

The **server-tags** list must include exactly one server tag or the keyword "all". It must not contain the **null** value.

15.4.8.2.16 remote-option-def4-get-all, remote-option-def6-get-all commands

These commands are used to fetch all DHCP option definitions from the database for the particular server or all servers. For example:

```
{
  "command": "remote-option-def6-get-all"
  "arguments": {
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "all" ]
  }
}
```

This command attempts to fetch all DHCPv6 option definitions associated with "all" servers. The **server-tags** list is mandatory for this command and it must include exactly one server tag or the keyword "all". It must not include the **null** value.

The following is the example response to this command:

```
{
  "result": 0,
  "text": "1 DHCPv6 option definition(s) found.",
  "arguments": {
    "option-defs": [
      {
        "name": "bar",
        "code": 1012,
        "space": "dhcp6",
        "type": "record",
        "array": true,
        "record-types": "ipv6-address, uint16",
        "encapsulate": "",
        "metadata": {
          "server-tags": [ "all" ]
        }
      }
    ]
  }
}
```



```
    }
  ],
  "count": 1
}
```

The response contains an option definition associated with all servers as indicated by the metadata.

15.4.8.2.17 remote-option-def4-set, remote-option-def6-set commands

These commands create a new DHCP option definition or replace an existing option definition in the database. The structure of the option definition information is the same as in the Kea configuration file (see Section 8.2.11 and Section 9.2.13). The following command creates the DHCPv4 option definition in the top level "dhcp4" option space and associates it with the "server1":

```
{
  "command": "remote-option-def4-set",
  "arguments": {
    "option-defs": [
      {
        "name": "foo",
        "code": 222,
        "type": "uint32",
        "array": false,
        "record-types": "",
        "space": "dhcp4",
        "encapsulate": ""
      }
    ],
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "server1" ]
  }
}
```

The **server-tags** list must include exactly one server tag or the keyword "all". It must not contain the **null** value.

15.4.8.2.18 remote-option4-global-del, remote-option6-global-del commands

These commands are used to delete a global DHCP option from the database. The option is identified by an option code and option space. For example:

```
{
  "command": "remote-option4-global-del",
  "arguments": {
    "options": [
      {
        "code": 5
        "space": "dhcp4"
      }
    ],
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "server1" ]
  }
}
```



The "dhcp4" is the top level option space where the standard DHCPv4 options belong. The **server-tags** is mandatory and it must include a single option tag or the keyword "all". If the explicit server tag is specified then this command attempts to delete a global option associated with this server. If there is no such option associated with the given server, no option is deleted. In order to delete the option associated with all servers, the keyword "all" must be specified.

15.4.8.2.19 remote-option4-global-get, remote-option6-global-get commands

These commands are used to fetch a global DHCP option from the database. The option is identified by the code and option space. The top level option spaces where DHCP standard options belong are called "dhcp4" and "dhcp6" for the DHCPv4 and DHCPv6 servers respectively.

The following command retrieves the IPv6 "DNS Servers" (code 23) option associated with all servers:

```
{
  "command": "remote-option6-global-get",
  "arguments": {
    "options": [
      {
        "code": 23,
        "space": "dhcp6"
      }
    ],
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "all" ]
  }
}
```

The **server-tags** is mandatory and it must include exactly one server tag or the keyword "all". It must not contain the **null** value.

15.4.8.2.20 remote-option4-global-get-all, remote-option6-global-get-all commands

These commands are used to fetch all global DHCP options from the configuration database for the particular server or for all servers. The following command fetches all global DHCPv4 options for the "server1":

```
{
  "command": "remote-option6-global-get-all",
  "arguments": {
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "server1" ]
  }
}
```

The **server-tags** list is mandatory for this command and it must contain exactly one server tag or a keyword "all". It must not contain the **null** value. The following is the example response to this command with a single option being associated with the "server1" returned:

```
{
  "result": 0,
  "text": "DHCPv4 options found.",
  "arguments": {
    "options": [
      {
        "name": "domain-name-servers",
        "code": 6,
        "space": "dhcp4",
        "csv-format": false,

```



```
        "data": "192.0.2.3",
        "metadata": {
            "server-tags": [ "server1" ]
        }
    },
    "count": 1
}
```

15.4.8.221 remote-option4-global-set, remote-option6-global-set command

These commands create a new global DHCP option or replace an existing option in the database. The structure of the option information is the same as in the Kea configuration file (see Section 8.2.10 and see Section 8.2.10). For example:

```
{
  "command": "remote-option6-global-set",
  "arguments": {
    "options": [
      {
        "name": "dns-servers",
        "data": "2001:db8:1::1"
      }
    ],
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "server1" ]
  }
}
```

The **server-tags** list is mandatory for this command and it must include exactly one server tag or the keyword "all". It must not include the **null** value. The command above associates the option with the "server1" server.

Note that specifying an option name instead of the option code only works reliably for the standard DHCP options. When specifying a value for the user defined DHCP option, the option code should be specified instead of the name. For example:

```
{
  "command": "remote-option6-global-set",
  "arguments": {
    "options": [
      {
        "code": 1,
        "space": "isc",
        "data": "2001:db8:1::1"
      }
    ],
    "server-tags": [ "server1" ]
  }
}
```

15.4.8.222 remote-subnet4-del-by-id, remote-subnet6-del-by-id command

This is the first variant of the commands used to delete an IPv4 or IPv6 subnet from the database. It uses subnet ID to identify the subnet. For example, in order to delete the IPv4 subnet with ID of 5:

```
{
  "command": "remote-subnet4-del-by-id",
  "arguments": {
```



```
    "subnets": [
      {
        "id": 5
      }
    ],
    "remote": {
      "type": "mysql"
    }
  }
}
```

The **server-tags** parameter must not be used with this command.

15.4.8.2.23 remote-subnet4-del-by-prefix, remote-subnet6-del-by-prefix commands

This is the second variant of the commands used to delete an IPv4 or IPv6 subnet from the database. It uses the subnet prefix to identify the subnet. For example:

```
{
  "command": "remote-subnet6-del-by-prefix",
  "arguments": {
    "subnets": [
      {
        "subnet": "2001:db8:1::/64"
      }
    ],
    "remote": {
      "type": "mysql"
    }
  }
}
```

The **server-tags** parameter must not be used with this command.

15.4.8.2.24 remote-subnet4-get-by-id, remote-subnet6-get-by-id commands

This is the first variant of the commands used to fetch an IPv4 or IPv6 subnet from the database. It uses a subnet ID to identify the subnet. For example:

```
{
  "command": "remote-subnet4-get-by-id",
  "arguments": {
    "subnets": [
      {
        "id": 5
      }
    ],
    "remote": {
      "type": "mysql"
    }
  }
}
```

The **server-tags** parameter must not be used with this command.

15.4.8.2.25 remote-subnet4-get-by-prefix, remote-subnet6-get-by-prefix command

This is the second variant of the commands used to fetch an IPv4 or IPv6 subnet from the database. It uses a subnet prefix to identify the subnet. For example:



```
{
  "command": "remote-subnet6-get-by-prefix",
  "arguments": {
    "subnets": [
      {
        "subnet": "2001:db8:1::/64"
      }
    ],
    "remote": {
      "type": "mysql"
    }
  }
}
```

The **server-tags** parameter must not be used with this command.

15.4.8.2.26 remote-subnet4-list, remote-subnet6-list commands

These commands are used to list all IPv4 or IPv6 subnets from the database for selected servers or all servers. The following command retrieves all servers to be used by the "server1" and "server2":

```
{
  "command": "remote-subnet4-list"
  "arguments": {
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ "server1", "server2" ]
  }
}
```

The **server-tags** parameter is mandatory and it contains one or more server tags. It may contain the keyword "all" to fetch the subnets associated with all servers. When the **server-tags** list contains the **null** value the returned response contains a list of unassigned subnets, i.e. the subnets which are associated with no servers. For example:

```
{
  "command": "remote-subnet4-list"
  "arguments": {
    "remote": {
      "type": "mysql"
    },
    "server-tags": [ null ]
  }
}
```

The example response to this command when non-null server tags are specified looks similar to this:

```
{
  "result": 0,
  "text": "2 IPv4 subnet(s) found.",
  "arguments": {
    "subnets": [
      {
        "id": 1,
        "subnet": "192.0.2.0/24",
        "shared-network-name": null,
        "metadata": {
          "server-tags": [ "server1", "server2" ]
        }
      }
    ],
  },
}
```



```
{
  "id": 2,
  "subnet": "192.0.3.0/24",
  "shared-network-name": null,
  "metadata": {
    "server-tags": [ "all" ]
  }
},
"count": 2
}
```

The returned information about each subnet is limited to subnet identifier, prefix and associated shared network name. In order to retrieve full information about the selected subnet use the **remote-subnet[46]-get-by-id** or **remote-subnet[46]-get-by-prefix**.

The example response above contains two subnets. One of the subnets is associated with both servers: "server1" and "server2". The second subnet is associated with all servers, thus it is also present in the configuration for the "server1" and "server2".

When listing unassigned subnets, the response will look similar to this:

```
{
  "result": 0,
  "text": "1 IPv4 subnet(s) found.",
  "arguments": {
    "subnets": [
      {
        "id": 3,
        "subnet": "192.0.4.0/24",
        "shared-network-name": null,
        "metadata": {
          "server-tags": [ null ]
        }
      }
    ],
    "count": 1
  }
}
```

The **null** value in the metadata indicates that the returned subnet is unassigned.

15.4.8.2.27 remote-subnet4-set, remote-subnet6-set commands

These commands are used to create a new IPv4 or IPv6 subnet or replace an existing subnet in the database. Setting the subnet also associates or disassociates the subnet with a shared network.

The structure of the subnet information is similar to the structure used in the configuration file (see Section 8.2 and Section 9.2). The subnet information conveyed in the **remote-subnet[46]-set** must include additional parameter **shared-network-name** which denotes whether the subnet belongs to a shared network or not.

Consider the following example:

```
{
  "command": "remote-subnet4-set",
  "arguments": {
    "subnets": [
      {
        "id": 5,
        "subnet": "192.0.2.0/24",
        "shared-network-name": "level3",
        "pools": [ { "pool": "192.0.2.100-192.0.2.200" } ],
        "option-data": [ {
```



```
        "name": "routers",
        "data": "192.0.2.1"
    } ]
}
],
"remote": {
    "type": "mysql"
},
"server-tags": [ "all" ]
}
}
```

It creates the subnet and associates the subnet with the "level3" shared network. The "level3" must be created with the **remote-network4-set** prior to creating the subnet.

If the created subnet must be global (not associated with any shared network), the **shared-network-name** must be explicitly set to **null**:

```
{
  "command": "remote-subnet4-set",
  "arguments": {
    "subnets": [
      {
        "id": 5,
        "subnet": "192.0.2.0/24",
        "shared-network-name": null,
        "pools": [ { "pool": "192.0.2.100-192.0.2.200" } ],
        "option-data": [ {
          "name": "routers",
          "data": "192.0.2.1"
        } ]
      }
    ]
  },
  "server-tags": [ "all" ]
}
}
```

The subnet created in the previous example is replaced with the new subnet having the same parameters but it becomes global.

The **shared-network-name** parameter is mandatory for the **remote-subnet4-set** command. The **server-tags** list is mandatory and it must include one or more server tags. As a result, the subnet is associated with all of the listed servers. It may also be associated with "all" servers connecting to the database when the keyword "all" is used as the server tag.

Note

Same as for other "set" commands, this command replaces the entire information about the particular subnet in the database, if the subnet information is already present. Therefore, when sending this command, make sure to always include all parameters that must be specified for the updated subnet instance. Any unspecified parameter will be marked as unspecified in the database, even if its value was present prior to sending the command.

15.4.9 ha: High Availability

This section describes the High Availability hooks library, which can be loaded on a pair of DHCPv4 or DHCPv6 servers to increase the reliability of the DHCP service in the event of an outage of one of the servers. This library was previously only available to ISC's paid subscribers, but is now part of the open source Kea, available to all users.

Note

This library may only be loaded by the **kea-dhcp4** or the **kea-dhcp6** process.



High Availability (HA) of the DHCP service is provided by running multiple cooperating server instances. If any of these instances becomes unavailable for any reason (DHCP software crash, Control Agent software crash, power outage, hardware failure), a surviving server instance can continue providing reliable service to clients. Many DHCP server implementations include the "DHCP Failover" protocol, whose most significant features are communication between the servers, partner failure detection, and lease synchronization between the servers. However, the DHCPv4 failover standardization process was never completed by the IETF. The DHCPv6 failover standard (RFC 8156) was published, but it is complex, difficult to use, has significant operational constraints, and is different than its v4 counterpart. Although it may be useful for some users to use a "standard" failover protocol, it seems that most Kea users are simply interested in a working solution which guarantees high availability of the DHCP service. Therefore, the Kea HA hook library derives major concepts from the DHCP Failover protocol but uses its own solutions for communication and configuration. It offers its own state machine, which greatly simplifies its implementation and generally fits better into Kea, and it provides the same features in both DHCPv4 and DHCPv6. This document intentionally uses the term "High Availability" rather than "Failover" to emphasize that it is not the Failover protocol implementation.

The following sections describe the configuration and operation of the Kea HA hook library.

15.4.9.1 Supported Configurations

The Kea HA hook library supports two configurations, also known as HA modes: load balancing and hot standby. In the load-balancing mode, two servers respond to DHCP requests. The load-balancing function is implemented as described in RFC 3074, with each server responding to half the received DHCP queries. When one of the servers allocates a lease for a client, it notifies the partner server over the control channel (RESTful API), so the partner can save the lease information in its own database. If the communication with the partner is unsuccessful, the DHCP query is dropped and the response is not returned to the DHCP client. If the lease update is successful, the response is returned to the DHCP client by the server which has allocated the lease. By exchanging lease updates, both servers get a copy of all leases allocated by the entire HA setup, and either server can be switched to handle the entire DHCP traffic if its partner becomes unavailable.

In the load-balancing configuration, one of the servers must be designated as "primary" and the other as "secondary." Functionally, there is no difference between the two during normal operation. This distinction is required when the two servers are started at (nearly) the same time and have to synchronize their lease databases. The primary server synchronizes the database first. The secondary server waits for the primary server to complete the lease database synchronization before it starts the synchronization.

In the hot-standby configuration, one of the servers is also designated as "primary" and the second as "secondary". However, during normal operation, the primary server is the only one that responds to DHCP requests. The secondary or standby server receives lease updates from the primary over the control channel; however, it does not respond to any DHCP queries as long as the primary is running or, more accurately, until the secondary considers the primary to be offline. If the secondary server detects the failure of the primary, it starts responding to all DHCP queries.

In the configurations described above, the primary, secondary, and standby are referred to as "active" servers, because they receive lease updates and can automatically react to the partner's failures by responding to the DHCP queries which would normally be handled by the partner. The HA hook library supports another server type/role: backup server. The use of a backup server is optional. They can be used in both load balancing and hot standby setup, in addition to the active servers. There is no limit on the number of backup servers in the HA setup; however, the presence of backup servers increases the latency of DHCP responses, because not only do active servers send lease updates to each other, but also to the backup servers.

15.4.9.2 Clocks on Active Servers

Synchronized clocks are essential for the HA setup to operate reliably. The servers share lease information via lease updates and during synchronization of the databases. The lease information includes the time when the lease has been allocated and when it expires. Some clock skew between the servers participating in the HA setup usually exists; this is acceptable as long as the clock skew is relatively low, compared to the lease lifetimes. However, if the clock skew becomes too high, the different lease expiration times on different servers may cause the HA system to malfunction. For example, one server may consider a lease to be expired when it is actually still valid. The lease reclamation process may remove a name associated with this lease from the DNS, causing problems when the client later attempts to renew the lease.

Each active server monitors the clock skew by comparing its current time with the time returned by its partner in response to the heartbeat command. This gives a good approximation of the clock skew, although it doesn't take into account the time between sending the response by the partner and receiving this response by the server which sent the heartbeat command. If the clock



skew exceeds 30 seconds, a warning log message is issued. The administrator may correct this problem by synchronizing the clocks (e.g. using NTP); the servers should notice the clock skew correction and stop issuing the warning

If the clock skew is not corrected and exceeds 60 seconds, the HA service on each of the servers is terminated, i.e. the state machine enters the **terminated** state. The servers will continue to respond to DHCP clients (as in the load-balancing or hot-standby mode), but will exchange neither lease updates nor heartbeats and their lease databases will diverge. In this case, the administrator should synchronize the clocks and restart the servers.

15.4.9.3 Server States

A DHCP server operating within an HA setup runs a state machine, and the state of the server can be retrieved by its peers using the **ha-heartbeat** command sent over the RESTful API. If the partner server doesn't respond to the **ha-heartbeat** command within the specified amount of time, the communication is considered interrupted and the server may (depending on the configuration) use additional measures (described later in this document) to verify that the partner is still operating. If it finds that the partner is not operating, the server transitions to the **partner-down** state to handle the entire DHCP traffic directed to the system.

In this case, the surviving server continues to send the **ha-heartbeat** command to detect when the partner wakes up. At that time, the partner synchronizes the lease database and when it is again ready to operate, the surviving server returns to normal operation, i.e. the **load-balancing** or **hot-standby** state.

The following is the list of all possible server states:

- **backup** - normal operation of the backup server. In this state it receives lease updates from the active servers.
- **hot-standby** - normal operation of the active server running in the hot-standby mode; both the primary and the standby server are in this state during their normal operation. The primary server responds to DHCP queries and sends lease updates to the standby server and to any backup servers that are present.
- **load-balancing** - normal operation of the active server running in the load-balancing mode; both the primary and the secondary server are in this state during their normal operation. Both servers respond to DHCP queries and send lease updates to each other and to any backup servers that are present.
- **partner-down** - an active server transitions to this state after detecting that its partner (another active server) is offline. The server does not transition to this state if only a backup server is unavailable. In the **partner-down** state the active server responds to all DHCP queries, including those queries which are normally handled by the server that is now unavailable.
- **ready** - an active server transitions to this state after synchronizing its lease database with an active partner. This state indicates to the partner - which may be in the **partner-down** state - that it should return to normal operation. If and when it does, the server in the **ready** state will also start normal operation.
- **syncing** - an active server transitions to this state to fetch leases from the active partner and update the local lease database. When in this state, the server issues the **dhcp-disable** command to disable the DHCP service of the partner from which the leases are fetched. The DHCP service is disabled for the maximum time of 60 seconds, after which it is automatically re-enabled, in case the syncing partner was unable to re-enable the service. If the synchronization is completed, the syncing server issues the **dhcp-enable** command to re-enable the DHCP service of its partner. The syncing operation is synchronous; the server waits for an answer from the partner and does nothing else while the lease synchronization takes place. A server that is configured not to synchronize the lease database with its partner, i.e. when the **sync-leases** configuration parameter is set to **false**, will never transition to this state. Instead, it will transition directly from the **waiting** state to the **ready** state.
- **terminated** - an active server transitions to this state when the High Availability hooks library is unable to further provide reliable service and a manual intervention of the administrator is required to correct the problem. Various issues with the HA setup may cause the server to transition to this state. While in this state, the server continues responding to DHCP clients based on the HA mode selected (load-balancing or hot-standby), but the lease updates are not exchanged and the heartbeats are not sent. Once a server has entered the "terminated" state, it will remain in this state until it is restarted. The administrator must correct the issue which caused this situation prior to restarting the server (e.g. synchronize clocks). Otherwise, the server will return to the "terminated" state once it finds that the issue persists.
- **waiting** - each started server instance enters this state. The backup server transitions directly from this state to the **backup** state. An active server sends a heartbeat to its partner to check its state; if the partner appears to be unavailable, the server transitions to the **partner-down** state. If the partner is available, the server transitions to the **syncing** or **ready** state, depending



on the setting of the **sync-leases** configuration parameter. If both servers appear to be in the **waiting** state (concurrent startup), the primary server transitions to the next state first. The secondary or standby server remains in the **waiting** state until the primary transitions to the **ready** state.

Note

Currently, restarting the HA service from the **terminated** state requires restarting the DHCP server or reloading its configuration.

Whether the server responds to the DHCP queries and which queries it responds to is a matter of the server's state, if no administrative action is performed to configure the server otherwise. The following table provides the default behavior for various states.

The **DHCP Server Scopes** denote what group of received DHCP queries the server responds to in the given state. An in-depth explanation of the scopes can be found below.

State	Server Type	DHCP Service	DHCP Service Scopes
backup	backup server	disabled	none
hot-standby	primary or standby (hot-standby mode)	enabled	HA_server1 if primary, none otherwise
load-balancing	primary or secondary (load-balancing mode)	enabled	HA_server1 or HA_server2
partner-down	active server	enabled	all scopes
ready	active server	disabled	none
syncing	active server	disabled	none
terminated	active server	enabled	same as in the load-balancing or hot-standby state
waiting	any server	disabled	none

Table 15.2: Default Behavior of the Server in Various HA States

The DHCP service scopes require some explanation. The HA configuration must specify a unique name for each server within the HA setup. This document uses the following convention within provided examples: **server1** for a primary server, **server2** for the secondary or standby server, and **server3** for the backup server. In real life any names can be used as long as they remain unique.

In the load-balancing mode there are two scopes named after the active servers: **HA_server1** and **HA_server2**. The DHCP queries load-balanced to **server1** belong to the **HA_server1** scope and the queries load-balanced to **server2** belong to the **HA_server2** scope. If either of the servers is in the **partner-down** state, the active partner is responsible for serving both scopes.

In the hot-standby mode, there is only one scope - **HA_server1** - because only **server1** is responding to DHCP queries. If that server becomes unavailable, **server2** becomes responsible for this scope.

The backup servers do not have their own scopes. In some cases they can be used to respond to queries belonging to the scopes of the active servers. Also, a server which is neither in the partner-down state nor in normal operation serves no scopes.

The scope names can be used to associate pools, subnets, and networks with certain servers, so only these servers can allocate addresses or prefixes from those pools, subnets, or networks. This is done via the client classification mechanism (see below).

15.4.9.4 Scope Transition in a Partner-Down Case

When one of the servers finds that its partner is unavailable, it starts serving clients from both its own scope and the scope of the unavailable partner. This is straightforward for new clients, i.e. those sending DHCPDISCOVER (DHCPv4) or Solicit (DHCPv6), because those requests are not sent to any particular server. The available server will respond to all such queries when it is in the **partner-down** state.



When a client renews a lease, it sends its DHCPREQUEST (DHCPv4) or Renew (DHCPv6) message directly to the server which has allocated the lease being renewed. If this server is no longer available, the client will get no response. In that case, the client continues to use its lease and attempts to renew until the rebind timer (T2) elapses. The client then enters the rebinding phase, in which it sends a DHCPREQUEST (DHCPv4) or Rebind (DHCPv6) message to any available server. The surviving server will receive the rebinding request and will typically extend the lifetime of the lease. The client then continues to contact that new server to renew its lease as appropriate.

If and when the other server once again becomes available, both active servers will eventually transition to the **load-balancing** or **hot-standby** state, in which they will again be responsible for their own scopes. Some clients belonging to the scope of the restarted server will try to renew their leases via the surviving server, but this server will not respond to them anymore; the client will eventually transition back to the correct server via the rebinding mechanism.

15.4.9.5 Load-Balancing Configuration

The following is the configuration snippet to enable high availability on the primary server within the load-balancing configuration. The same configuration should be applied on the secondary and backup servers, with the only difference that **this-server-name** should be set to **server2** and **server3** on those servers, respectively.

```
{
  "Dhcp4": {
    ...

    "hooks-libraries": [
      {
        "library": "/usr/lib/kea/hooks/libdhcp_lease_cmds.so",
        "parameters": { }
      },
      {
        "library": "/usr/lib/kea/hooks/libdhcp_ha.so",
        "parameters": {
          "high-availability": [ {
            "this-server-name": "server1",
            "mode": "load-balancing",
            "heartbeat-delay": 10000,
            "max-response-delay": 10000,
            "max-ack-delay": 5000,
            "max-unacked-clients": 5,
            "peers": [
              {
                "name": "server1",
                "url": "http://192.168.56.33:8080/",
                "role": "primary",
                "auto-failover": true
              },
              {
                "name": "server2",
                "url": "http://192.168.56.66:8080/",
                "role": "secondary",
                "auto-failover": true
              },
              {
                "name": "server3",
                "url": "http://192.168.56.99:8080/",
                "role": "backup",
                "auto-failover": false
              }
            ]
          }
        ]
      }
    ]
  }
}
```



```
],
  "subnet4": [
    {
      "subnet": "192.0.3.0/24",
      "pools": [
        {
          "pool": "192.0.3.100 - 192.0.3.150",
          "client-class": "HA_server1"
        },
        {
          "pool": "192.0.3.200 - 192.0.3.250",
          "client-class": "HA_server2"
        }
      ]
    },
    {
      "option-data": [
        {
          "name": "routers",
          "data": "192.0.3.1"
        }
      ]
    },
    {
      "relay": { "ip-address": "10.1.2.3" }
    }
  ],
  ...
}
```

Two hook libraries must be loaded to enable HA: `libdhcp_lease_cmds.so` and `libdhcp_ha.so`. The latter implements the HA feature, while the former enables control commands required by HA to fetch and manipulate leases on the remote servers. In the example provided above, it is assumed that Kea libraries are installed in the `/usr/lib` directory. If Kea is not installed in the `/usr` directory, the hook libraries locations must be updated accordingly.

The HA configuration is specified within the scope of `libdhcp_ha.so`. Note that the top-level parameter **high-availability** is a list, even though it currently contains only one entry.

The following are the global parameters which control the server's behavior with respect to HA:

- **this-server-name** - is a unique identifier of the server within this HA setup. It must match with one of the servers specified within the **peers** list.
- **mode** - specifies an HA mode of operation. Currently supported modes are **load-balancing** and **hot-standby**.
- **heartbeat-delay** - specifies a duration in milliseconds between sending the last heartbeat (or other command sent to the partner) and the next heartbeat. The heartbeats are sent periodically to gather the status of the partner and to verify whether the partner is still operating. The default value of this parameter is 10000 ms.
- **max-response-delay** - specifies a duration in milliseconds since the last successful communication with the partner, after which the server assumes that communication with the partner is interrupted. This duration should be greater than the **heartbeat-delay**. Usually it is greater than the duration of multiple **heartbeat-delay** values. When the server detects that communication is interrupted, it may transition to the **partner-down** state (when **max-unacked-clients** is 0) or trigger the failure- detection procedure using the values of the two parameters below. The default value of this parameter is 60000.
- **max-ack-delay** - is one of the parameters controlling partner failure-detection. When communication with the partner is interrupted, the server examines the values of the **secs** field (DHCPv4) or **Elapsed Time** option (DHCPv6), which denote how long the DHCP client has been trying to communicate with the DHCP server. This parameter specifies the maximum time in milliseconds for the client to try to communicate with the DHCP server, after which this server assumes that the client failed to communicate with the DHCP server (is "unacked"). The default value of this parameter is 10000.



- **max-unacked-clients** - specifies how many "unacked" clients are allowed (see **max-ack-delay**) before this server assumes that the partner is offline and transitions to the **partner-down** state. The special value of 0 is allowed for this parameter, which disables the failure-detection mechanism. In this case, a server that can't communicate with its partner over the control channel assumes that the partner server is down and transitions to the **partner-down** state immediately. The default value of this parameter is 10.

The values of **max-ack-delay** and **max-unacked-clients** must be selected carefully, taking into account the specifics of the network in which the DHCP servers are operating. Note that the server in question may not respond to some DHCP clients because these clients are not to be serviced by this server according to administrative policy. The server may also drop malformed queries from clients. Therefore, selecting too low a value for the **max-unacked-clients** parameter may result in a transition to the **partner-down** state even though the partner is still operating. On the other hand, selecting too high a value may result in never transitioning to the **partner-down** state if the DHCP traffic in the network is very low (e.g. nighttime), because the number of distinct clients trying to communicate with the server could be lower than the **max-unacked-clients** setting.

In some cases it may be useful to disable the failure-detection mechanism altogether, if the servers are located very close to each other and network partitioning is unlikely, i.e. failure to respond to heartbeats is only possible when the partner is offline. In such cases, set the **max-unacked-clients** to 0.

The **peers** parameter contains a list of servers within this HA setup. This configuration must contain at least one primary and one secondary server. It may also contain an unlimited number of backup servers. In this example, there is one backup server which receives lease updates from the active servers.

These are the parameters specified for each of the peers within this list:

- **name** - specifies a unique name for the server.
- **url** - specifies the URL to be used to contact this server over the control channel. Other servers use this URL to send control commands to that server.
- **role** - denotes the role of the server in the HA setup. The following roles are supported in the load-balancing configuration: **primary**, **secondary**, and **backup**. There must be exactly one primary and one secondary server in the load-balancing setup.
- **auto-failover** - a boolean value which denotes whether a server detecting a partner's failure should automatically start serving the partner's clients. The default value of this parameter is true.

In our example configuration, both active servers can allocate leases from the subnet "192.0.3.0/24". This subnet contains two address pools: "192.0.3.100 - 192.0.3.150" and "192.0.3.200 - 192.0.3.250", which are associated with HA server scopes using client classification. When **server1** processes a DHCP query, it uses the first pool for lease allocation. Conversely, when **server2** processes a DHCP query it uses the second pool. When either of the servers is in the **partner-down** state, it can serve leases from both pools and it selects the pool which is appropriate for the received query. In other words, if the query would normally be processed by **server2** but this server is not available, **server1** will allocate the lease from the pool of "192.0.3.200 - 192.0.3.250".

15.4.9.6 Load Balancing with Advanced Classification

In the previous section, we provided an example of a load-balancing configuration with client classification limited to the **HA_server1** and **HA_server2** classes, which are dynamically assigned to the received DHCP queries. In many cases, HA will be needed in deployments which already use some other client classification.

Suppose there is a system which classifies devices into two groups: phones and laptops, based on some classification criteria specified in Kea configuration file. Both types of devices are allocated leases from different address pools. Introducing HA in the load-balancing mode results in a further split of each of those pools, as each server allocates leases for some phones and some laptops. This requires each of the existing pools to be split between **HA_server1** and **HA_server2**, so we end up with the following classes:

- phones_server1
- laptops_server1
- phones_server2



- laptops_server2

The corresponding server configuration using advanced classification (and **member** expression) is provided below. For brevity's sake, the HA hook library configuration has been removed from this example.

```
{
  "Dhcp4": {
    "client-classes": [
      {
        "name": "phones",
        "test": "substring(option[60].hex,0,6) == 'Aastra'",
      },
      {
        "name": "laptops",
        "test": "not member('phones')"
      },
      {
        "name": "phones_server1",
        "test": "member('phones') and member('HA_server1')"
      },
      {
        "name": "phones_server2",
        "test": "member('phones') and member('HA_server2')"
      },
      {
        "name": "laptops_server1",
        "test": "member('laptops') and member('HA_server1')"
      },
      {
        "name": "laptops_server2",
        "test": "member('laptops') and member('HA_server2')"
      }
    ],
    "hooks-libraries": [
      {
        "library": "/usr/lib/kea/hooks/libdhcp_lease_cmds.so",
        "parameters": { }
      },
      {
        "library": "/usr/lib/kea/hooks/libdhcp_ha.so",
        "parameters": {
          "high-availability": [ {
            ...
          } ]
        }
      }
    ],
    "subnet4": [
      {
        "subnet": "192.0.3.0/24",
        "pools": [
          {
            "pool": "192.0.3.100 - 192.0.3.125",
            "client-class": "phones_server1"
          },
          {
            "pool": "192.0.3.126 - 192.0.3.150",
```



```
        "client-class": "laptops_server1"
      },
      {
        "pool": "192.0.3.200 - 192.0.3.225",
        "client-class": "phones_server2"
      },
      {
        "pool": "192.0.3.226 - 192.0.3.250",
        "client-class": "laptops_server2"
      }
    ],
    "option-data": [
      {
        "name": "routers",
        "data": "192.0.3.1"
      }
    ],
    "relay": { "ip-address": "10.1.2.3" }
  },
  ...
}
```

The configuration provided above splits the address range into four pools: two pools dedicated to server1 and two to server2. Each server can assign leases to both phones and laptops. Both groups of devices are assigned addresses from different pools. The **HA_server1** and **HA_server2** classes are built-in (see Section 14.2) and do not need to be declared. They are assigned dynamically by the HA hook library as a result of the load-balancing algorithm. **phones_*** and **laptop_*** evaluate to "true" when the query belongs to a given combination of other classes, e.g. **HA_server1** and **phones**. The pool is selected accordingly as a result of such an evaluation.

Consult Chapter 14 for details on how to use the **member** expression and class dependencies.

15.4.9.7 Hot-Standby Configuration

The following is an example configuration of the primary server in the hot-standby configuration:

```
{
  "Dhcp4": {
    ...

    "hooks-libraries": [
      {
        "library": "/usr/lib/kea/hooks/libdhcp_lease_cmds.so",
        "parameters": { }
      },
      {
        "library": "/usr/lib/kea/hooks/libdhcp_ha.so",
        "parameters": {
          "high-availability": [ {
            "this-server-name": "server1",
            "mode": "hot-standby",
            "heartbeat-delay": 10000,
            "max-response-delay": 10000,
            "max-ack-delay": 5000,
          }
        ]
      }
    ]
  }
}
```



```
        "max-unacked-clients": 5,
        "peers": [
            {
                "name": "server1",
                "url": "http://192.168.56.33:8080/",
                "role": "primary",
                "auto-failover": true
            },
            {
                "name": "server2",
                "url": "http://192.168.56.66:8080/",
                "role": "standby",
                "auto-failover": true
            },
            {
                "name": "server3",
                "url": "http://192.168.56.99:8080/",
                "role": "backup",
                "auto-failover": false
            }
        ]
    } ]
}
],
"subnet4": [
    {
        "subnet": "192.0.3.0/24",
        "pools": [
            {
                "pool": "192.0.3.100 - 192.0.3.250",
                "client-class": "HA_server1"
            }
        ],
        "option-data": [
            {
                "name": "routers",
                "data": "192.0.3.1"
            }
        ],
        "relay": { "ip-address": "10.1.2.3" }
    }
],
...
}
}
```

This configuration is very similar to the load-balancing configuration described in Section [15.4.9.5](#), with a few notable differences.

The **mode** is now set to **hot-standby**, in which only one server responds to DHCP clients. If the primary server is online, it responds to all DHCP queries. The **standby** server takes over all DHCP traffic if it discovers that the primary is unavailable.

In this mode, the non-primary active server is called **standby** and that is its role.

Finally, because there is always one server responding to DHCP queries, there is only one scope - **HA_server1** - in use within pools definitions. In fact, the **client-class** parameter could be removed from this configuration without harm, because there can



be no conflicts in lease allocations by different servers as they do not allocate leases concurrently. The **client-class** remains in this example mostly for demonstration purposes, to highlight the differences between the hot-standby and load-balancing modes of operation.

15.4.9.8 Lease Information Sharing

An HA-enabled server informs its active partner about allocated or renewed leases by sending appropriate control commands, and the partner updates the lease information in its own database. When the server starts up for the first time or recovers after a failure, it synchronizes its lease database with its partner. These two mechanisms guarantee consistency of the lease information between the servers and allow the designation of one of the servers to handle the entire DHCP traffic load if the other server becomes unavailable.

In some cases, though, it is desirable to disable lease updates and/or database synchronization between the active servers, if the exchange of information about the allocated leases is performed using some other mechanism. Kea supports various database types that can be used to store leases, including MySQL, Postgres, and Cassandra. Those databases include built-in solutions for data replication which are often used by Kea administrators to provide redundancy.

The HA hook library supports such scenarios by disabling lease updates over the control channel and/or lease database synchronization, leaving the server to rely on the database replication mechanism. This is controlled by the two boolean parameters **send-lease-updates** and **sync-leases**, whose values default to true:

```
{
  "Dhcp4": {
    ...
    "hooks-libraries": [
      {
        "library": "/usr/lib/kea/hooks/libdhcp_lease_cmds.so",
        "parameters": { }
      },
      {
        "library": "/usr/lib/kea/hooks/libdhcp_ha.so",
        "parameters": {
          "high-availability": [ {
            "this-server-name": "server1",
            "mode": "load-balancing",
            "send-lease-updates": false,
            "sync-leases": false,
            "peers": [
              {
                "name": "server1",
                "url": "http://192.168.56.33:8080/",
                "role": "primary"
              },
              {
                "name": "server2",
                "url": "http://192.168.56.66:8080/",
                "role": "secondary"
              }
            ]
          } ]
        } ]
      }
    ],
    ...
  }
}
```



In the most typical use case, both parameters are set to the same value, i.e. both are **false** if database replication is in use, or both are **true** otherwise. Introducing two separate parameters to control lease updates and lease-database synchronization is aimed at possible special use cases; for example, when synchronization is performed by copying a lease file (therefore **sync-leases** is set to **false**), but lease updates should be conducted as usual (**send-lease-updates** is set to **true**). It should be noted that Kea does not natively support such use cases, but users may develop their own scripts and tools around Kea to provide such mechanisms. The HA hooks library configuration is designed to maximize flexibility of administration.

15.4.9.9 Controlling Lease-Page Size Limit

An HA-enabled server initiates synchronization of the lease database after downtime or upon receiving the **ha-sync** command. The server uses commands described in Section 15.4.5.1.4 to fetch leases from its partner server (lease queries). The size of the results page (the maximum number of leases to be returned in a single response to one of these commands) can be controlled via HA hooks library configuration. Increasing the page size decreases the number of lease queries sent to the partner server, but it causes the partner server to generate larger responses, which lengthens transmission time as well as increases memory and CPU utilization on both servers. Decreasing the page size helps to decrease resource utilization, but requires more lease queries to be issued to fetch the entire lease database.

The default value of the **sync-page-limit** command controlling the page size is 10000. This means that the entire lease database can be fetched with a single command if the size of the database is equal to or less than 10000 lines.

15.4.9.10 Discussion About Timeouts

In deployments with a large number of clients connected to the network, lease-database synchronization after a server failure may be a time-consuming operation. The synchronizing server must gather all leases from its partner, which yields a large response over the RESTful interface. The server receives leases using the paging mechanism described in Section 15.4.9.9. Before the page of leases is fetched, the synchronizing server sends a **dhcp-disable** command to disable the DHCP service on the partner server. If the service is already disabled, this command will reset the timeout for the DHCP service being disabled. This timeout value is by default set to 60 seconds. If fetching a single page of leases takes longer than the specified time, the partner server will assume that the synchronizing server died and will resume its DHCP service. The connection of the synchronizing server with its partner is also protected by the timeout. If the synchronization of a single page of leases takes longer than the specified time, the synchronizing server terminates the connection and the synchronization fails. Both timeout values are controlled by a single configuration parameter: **sync-timeout**. The following configuration snippet demonstrates how to modify the timeout for automatic re-enabling of the DHCP service on the partner server and how to increase the timeout for fetching a single page of leases from 60 seconds to 90 seconds:

```
{
"Dhcp4": {
    ...
    "hooks-libraries": [
        {
            "library": "/usr/lib/kea/hooks/libdhcp_lease_cmds.so",
            "parameters": { }
        },
        {
            "library": "/usr/lib/kea/hooks/libdhcp_ha.so",
            "parameters": {
                "high-availability": [ {
                    "this-server-name": "server1",
                    "mode": "load-balancing",
                    "sync-timeout": 90000,
                    "peers": [
                        {
                            "name": "server1",
                            "url": "http://192.168.56.33:8080/",
                            "role": "primary"
                        }
                    ]
                }
            ]
        }
    ]
}
```



```
        {
            "name": "server2",
            "url": "http://192.168.56.66:8080/",
            "role": "secondary"
        }
    ]
} ]
}
},
],
...
}
```

It is important to note that extending this **sync-timeout** value may sometimes be insufficient to prevent issues with timeouts during lease-database synchronization. The control commands travel via the Control Agent, which also monitors incoming (with a synchronizing server) and outgoing (with a DHCP server) connections for timeouts. The DHCP server also monitors the connection from the Control Agent for timeouts. Those timeouts cannot currently be modified via configuration; extending these timeouts is only possible by modifying them in the Kea code and recompiling the server. The relevant constants are located in the Kea source at: `src/lib/config/timeouts.h`.

15.4.9.11 Pausing HA State Machine

The high-availability state machine includes many different states described in detail in Section 15.4.9.3. The server enters each state when certain conditions are met, most often taking into account the partner server's state. In some states the server performs specific actions, e.g. synchronization of the lease database in the **syncing** state or responding to DHCP queries according to the configured mode of operation in the **load-balancing** and **hot-standby** states.

By default, transitions between the states are performed automatically and the server administrator has no direct control when the transitions take place; in most cases, the administrator doesn't need such control. In some situations, however, the administrator may want to "pause" the HA state machine in a selected state to perform some additional administrative actions before the server transitions to the next state.

Consider a server failure which results in the loss of the entire lease database. Typically, the server will rebuild its lease database when it enters the **syncing** state by querying the partner server for leases, but it is possible that the partner was also experiencing a failure and lacks lease information. In this case, it may be required to reconstruct lease databases on both servers from some external source, e.g. a backup server. If the lease database is to be reconstructed via RESTful API, the servers should be started in the initial, i.e. **waiting**, state and remain in this state while leases are being added. In particular, the servers should not attempt to synchronize their lease databases nor start serving DHCP clients.

The HA hooks library provides configuration parameters and a command to control when the HA state machine should be paused and resumed. The following configuration causes the HA state machine to pause in the **waiting** state after server startup.

```
"Dhcp4": {
    ...
    "hooks-libraries": [
        {
            "library": "/usr/lib/kea/hooks/libdhcp_lease_cmds.so",
            "parameters": { }
        },
        {
            "library": "/usr/lib/kea/hooks/libdhcp_ha.so",
            "parameters": {
                "high-availability": [ {
                    "this-server-name": "server1",
                    "mode": "load-balancing",
                    "peers": [
```



```
        {
            "name": "server1",
            "url": "http://192.168.56.33:8080/",
            "role": "primary"
        },
        {
            "name": "server2",
            "url": "http://192.168.56.66:8080/",
            "role": "secondary"
        }
    ],
    "state-machine": {
        "states": [
            {
                "state": "waiting",
                "pause": "once"
            }
        ]
    }
} ] ]
}
},
],
...
}
```

The **pause** parameter value **once** denotes that the state machine should be paused upon the first transition to the **waiting** state; later transitions to this state will not cause the state machine to pause. Two other supported values of the **pause** parameter are: **always** and **never**. The latter is the default value for each state, which instructs the server never to pause the state machine.

In order to "unpause" the state machine, the **ha-continue** command must be sent to the paused server. This command does not take any arguments. See Section 15.4.9.13 for details about commands specific to the HA hooks library.

It is possible to configure the state machine to pause in more than one state. Consider the following configuration:

```
"Dhcp4": {
    ...
    "hooks-libraries": [
        {
            "library": "/usr/lib/kea/hooks/libdhcp_lease_cmds.so",
            "parameters": { }
        },
        {
            "library": "/usr/lib/kea/hooks/libdhcp_ha.so",
            "parameters": {
                "high-availability": [ {
                    "this-server-name": "server1",
                    "mode": "load-balancing",
                    "peers": [
                        {
                            "name": "server1",
                            "url": "http://192.168.56.33:8080/",
                            "role": "primary"
                        },
                        {
                            "name": "server2",
                            "url": "http://192.168.56.66:8080/",
                            "role": "secondary"
                        }
                    ]
                }
            ]
        }
    ]
}
```



```
    ],
    "state-machine": {
      "states": [
        {
          "state": "ready",
          "pause": "always"
        },
        {
          "state": "partner-down",
          "pause": "once"
        }
      ]
    }
  } ]
}
...
}
```

This configuration instructs the server to pause the state machine every time it transitions to the **ready** state and upon the first transition to the **partner-down** state.

Refer to Section [15.4.9.3](#) for a complete list of server states. The state machine can be paused in any of the supported states; however, it is not practical for the **backup** and **terminated** states because the server never transitions out of these states anyway.

Note

In the **syncing** state the server is paused before it makes an attempt to synchronize the lease database with a partner. To pause the state machine after lease-database synchronization, use the **ready** state instead.

Note

The state of the HA state machine depends on the state of the cooperating server. Therefore, it must be taken into account that pausing the state machine of one server may affect the operation of the partner server. For example: if the primary server is paused in the **waiting** state, the partner server will also remain in the **waiting** state until the state machine of the primary server is resumed and that server transitions to the **ready** state.

15.4.9.12 Control Agent Configuration

Chapter [7](#) describes in detail the Kea daemon, which provides a RESTful interface to control Kea servers. The same functionality is used by the High Availability hook library to establish communication between the HA peers. Therefore, the HA library requires that the Control Agent (CA) be started for each DHCP instance within the HA setup. If the Control Agent is not started, the peers will not be able to communicate with the particular DHCP server (even if the DHCP server itself is online) and may eventually consider this server to be offline.

The following is an example configuration for the CA running on the same machine as the primary server. This configuration is valid for both the load-balancing and the hot-standby cases presented in previous sections.

```
{
"Control-agent": {
  "http-host": "192.168.56.33",
  "http-port": 8080,

  "control-sockets": {
    "dhcp4": {
      "socket-type": "unix",
```



```
        "socket-name": "/tmp/kea-dhcp4-ctrl.sock"
    },
    "dhcp6": {
        "socket-type": "unix",
        "socket-name": "/tmp/kea-dhcp6-ctrl.sock"
    }
}
}
```

15.4.9.13 Control Commands for High Availability

Even though the HA hook library is designed to automatically resolve issues with DHCP service interruptions by redirecting the DHCP traffic to a surviving server and synchronizing the lease database when required, it may be useful for the administrator to have more control over the server behavior. In particular, it may be useful to be able to trigger lease-database synchronization on demand. It may also be useful to manually set the HA scopes that are being served.

Note that the backup server can sometimes be used to handle DHCP traffic if both active servers are down. The backup servers do not perform failover function automatically. Thus, in order to use the backup server to respond to DHCP queries, the server administrator must enable this function manually.

The following sections describe commands supported by the HA hook library which are available for the administrator.

15.4.9.13.1 ha-sync Command

The **ha-sync** command instructs the server to synchronize its local lease database with the selected peer. The server fetches all leases from the peer and updates those locally stored leases which are older than those fetched. It also creates new leases when any of those fetched do not exist in the local database. All leases that are not returned by the peer but are in the local database are preserved. The database synchronization is unidirectional; only the database on the server to which the command has been sent is updated. In order to synchronize the peer's database a separate **ha-sync** has to be issued to that peer.

Database synchronization may be triggered for both active and backup server types. The **ha-sync** command has the following structure (DHCPv4 server case):

```
{
  "command": "ha-sync",
  "service": [ "dhcp4 "],
  "arguments": {
    "server-name": "server2",
    "max-period": 60
  }
}
```

When the server receives this command it first disables the DHCP service of the server from which it will be fetching leases, by sending the **dhcp-disable** command to that server. The **max-period** parameter specifies the maximum duration (in seconds) for which the DHCP service should be disabled. If the DHCP service is successfully disabled, the synchronizing server will fetch leases from the remote server by issuing one or more **lease4-get-page** commands. When the lease-database synchronization is complete, the synchronizing server sends the **dhcp-enable** command to the peer to re-enable its DHCP service.

The **max-period** value should be sufficiently long to guarantee that it doesn't elapse before the synchronization is completed. Otherwise, the DHCP server will automatically enable its DHCP function while the synchronization is still in progress. If the DHCP server subsequently allocates any leases during the synchronization, those new (or updated) leases will not be fetched by the synchronizing server, leading to database inconsistencies.

15.4.9.13.2 ha-scopes Command

This command allows modification of the HA scopes that the server is serving. Consult Section [15.4.9.5](#) and Section [15.4.9.7](#) to learn what scopes are available for different HA modes of operation. The **ha-scopes** command has the following structure (DHCPv4 server case):



```
{
  "command": "ha-scopes",
  "service": [ "dhcp4" ],
  "arguments": {
    "scopes": [ "HA_server1", "HA_server2" ]
  }
}
```

This command configures the server to handle traffic from both **HA_server1** and **HA_server2** scopes. In order to disable all scopes specify an empty list:

```
{
  "command": "ha-scopes",
  "service": [ "dhcp4" ],
  "arguments": {
    "scopes": [ ]
  }
}
```

15.4.9.13.3 ha-continue Command

This command is used to resume the operation of the paused HA state machine, as described in Section 15.4.9.11. It takes no arguments, so the command structure is as simple as:

```
{
  "command": "ha-continue"
}
```

15.4.10 stat_cmds: Supplemental Statistics Commands

This library provides additional commands for retrieving lease statistics from Kea DHCP servers. These commands were added to address an issue with obtaining accurate lease statistics in deployments running multiple Kea servers that use a shared lease backend. The in-memory statistics kept by individual servers only track lease changes made by that server; thus, in a deployment with multiple servers (e.g. two kea-dhcp6 servers using the same PostgreSQL database for lease storage), these statistics are incomplete. The MySQL and PostgreSQL backends in Kea track lease allocation changes as they occur via database triggers. Additionally, all four lease backends were extended to support retrieving lease statistics for all subnets, a single subnet, or a range of subnets. Finally, this library was constructed to provide commands for retrieving these statistics.

Note

This library may only be loaded by the **kea-dhcp4** or the **kea-dhcp6** process.

The commands currently provided by this library are:

- **stat-lease4-get** - fetches DHCPv4 lease statistics
- **stat-lease6-get** - fetches DHCPv6 lease statistics

The Stat commands library is part of the open source code and is available to every Kea user.

All commands use JSON syntax and can be issued directly to the servers via either the control channel (see Chapter 17) or the Control Agent (see Chapter 7).

This library may be loaded by both kea-dhcp4 and kea-dhcp6 servers. It is loaded in the same way as other libraries and currently has no parameters:



```
"Dhcp6": {
  "hooks-libraries": [
    {
      "library": "/path/libdhcp_stat_cmds.so"
    }
    ...
  ]
}
```

In a deployment with multiple Kea DHCP servers sharing a common lease storage, it may be loaded by any or all of the servers. However, one thing to keep in mind is that a server's response to a `stat-lease{4/6}-get` command will only contain data for subnets known to that server. In other words, if a subnet does not appear in a server's configuration, Kea will not retrieve statistics for it.

15.4.10.1 `stat-lease4-get`, `stat-lease6-get` Commands

The `stat-lease4-get` and `stat-lease6-get` commands fetch lease statistics for a range of known subnets. The range of subnets is determined through the use of optional command input parameters:

- **subnet-id** - The ID of the subnet for which lease statistics should be fetched. Use this to get statistics for a single subnet. If the subnet does not exist, the command result code is 3 (i.e. `CONTROL_RESULT_EMPTY`).
- **subnet-range** - A pair of subnet IDs which describe an inclusive range of subnets for which statistics should be retrieved. The range may include one or more IDs that correspond to no subnet: in this case, the command will only output lease statistics for those that exist. However, if the range does not include any known subnets, the command result code is 3 (i.e. `CONTROL_RESULT_EMPTY`).
 - **first-subnet-id** - The ID of the first subnet in the range.
 - **last-subnet-id** - The ID of the last subnet in the range.

The use of `subnet-id` and `subnet-range` are mutually exclusive. If no parameters are given, the result will contain data for all known subnets. Note that in configurations with large numbers of subnets, this can result in a large response.

The following command fetches lease statistics for all known subnets from a `kea-dhcp4` server:

```
{
  "command": "stat-lease4-get"
}
```

The following command fetches lease statistics for subnet ID 10 from a `kea-dhcp6` server:

```
{
  "command": "stat-lease6-get",
  "arguments": {
    "subnet-id" : 10
  }
}
```

The following command fetches lease statistics for all subnets with IDs in the range 10 through 50 from a `kea-dhcp4` server:

```
{
  "command": "stat-lease4-get",
  "arguments": {
    "subnet-range" {
      "first-subnet-id": 10,
      "last-subnet-id": 50,
    }
  }
}
```



The response to either command will contain three elements:

- **result** - a numeric value indicating the outcome of the command where:
 - **0** - the command was successful;
 - **1** - an error occurred, an explanation will be the "text" element;
 - **2** - the fetch found no matching data
- **text** - an explanation of the command outcome. When the command succeeds it will contain the command name along with the number of rows returned.
- **arguments** - a map containing the data returned by the command as the element "result-set", which is patterned after SQL statement responses:
 - **columns** - a list of text column labels The columns returned for DHCPv4 are:
 - * **subnet-id** - ID of the subnet.
 - * **total-addresses** - total number of addresses available for DHCPv4 management in the subnet. In other words, this is the sum of all addresses in all the configured pools in the subnet.
 - * **assigned-addresses** - number of addresses in the subnet that are currently assigned to a client.
 - * **declined-addresses** - number of addresses in the subnet that are currently declined and are thus unavailable for assignment.
 - The columns returned for DHCPv6 are:
 - * **subnet-id** - ID of the subnet.
 - * **total-nas** - number of NA addresses available for DHCPv6 management in the subnet. In other words, this is the sum of all the NA addresses in the all configured NA pools in the subnet.
 - * **assigned-nas** - number of NA addresses in a the subnet that are currently assigned to a client.
 - * **declined-nas** - number of NA addresses that are currently declined and are thus unavailable for assignment.
 - * **total-pds** - total number of prefixes available of DHCPv6 management in the subnet. In other words, this is the sum of all prefixes in all the configured prefix pools in the subnet.
 - * **assigned-pds** - number of prefixes in the subnet that are currently assigned to a client.
 - **rows** - a list of rows, one per subnet ID. Each row contains a data value corresponding to and in the same order as each column listed in "columns" for a given subnet.
 - **timestamp** - textual date and time the data was fetched, expressed as GMT.

The response to a DHCPv4 command might look as follows:

```
{
  "result": 0,
  "text": "stat-lease4-get: 2 rows found",
  "arguments": {
    "result-set": {
      "columns": [ "subnet-id", "total-addresses", "assigned-addresses", "declined- ↵
        addresses" ]
      "rows": [
        [ 10, 256, 111, 0 ],
        [ 20, 4098, 2034, 4 ]
      ],
      "timestamp": "2018-05-04 15:03:37.000000"
    }
  }
}
```

The response to a DHCPv6 command might look as follows (subnet 10 has no prefix pools, subnet 20 has no NA pools, and subnet 30 has both NA and PD pools):



```
{
  "result": 0,
  "text": "stat-lease6-get: 2 rows found",
  "arguments": {
    "result-set": {
      "columns": [ "subnet-id", "total-nas", "assigned-nas", "declined-nas", "total-pds", ←
        "assigned-pds" ]
      "rows": [
        [ 10, 4096, 2400, 3, 0, 0 ],
        [ 20, 0, 0, 0, 1048, 233 ]
        [ 30, 256, 60, 0, 1048, 15 ]
      ],
      "timestamp": "2018-05-04 15:03:37.000000"
    }
  }
}
```

15.4.11 radius: RADIUS Server Support

The RADIUS hook library allows Kea to interact with two types of RADIUS servers: access and accounting. Although the most common DHCP and RADIUS integration is done on the DHCP relay-agent level (DHCP clients send DHCP packets to DHCP relays; those relays contact the RADIUS server and depending on the response either send the packet to the DHCP server or drop it), it does require DHCP relay hardware to support RADIUS communication. Also, even if the relay has the necessary support, it is often not flexible enough to send and receive additional RADIUS attributes. As such, the alternative looks more appealing: to extend the DHCP server to talk to RADIUS directly. That is the goal this library intends to fulfill.

Note

This library may only be loaded by the **kea-dhcp4** or the **kea-dhcp6** process.

The major feature of the hook library is the ability to use RADIUS authorization. When a DHCP packet is received, the Kea server sends an Access-Request to the RADIUS server and waits for a response. The server then sends back either an Access-Accept with specific client attributes, or an Access-Reject. There are two cases supported here: first, the Access-Accept includes a Framed-IP-Address (for DHCPv4) or Framed-IPv6-Address (for DHCPv6), which will be interpreted by Kea as an instruction to assign that specified IPv4 or IPv6 address. This effectively means RADIUS can act as an address-reservation database.

The second case supported is the ability to assign clients to specific pools based on a RADIUS response. In this case, the RADIUS server sends back an Access-Accept with Framed-Pool (IPv4) or Framed-IPv6-Pool (IPv6). In both cases, Kea interprets those attributes as client classes. With the addition of the ability to limit access to pools to specific classes (see Section 14.7), RADIUS can be used to force the client to be assigned a dynamic address from a specific pool. Furthermore, the same mechanism can be used to control what kind of options the client will get if there are DHCP options specified for a particular class.

15.4.11.1 Compilation and Installation of the RADIUS Hook

The following section describes how to compile and install the software on CentOS 7.0. Other systems may differ slightly.

STEP 1: Install dependencies

Several tools are needed to build dependencies and Kea itself. The following commands should install them:

```
$ sudo rpm -Uvh https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
$ sudo yum install gcc-g++ openssl-devel log4cplus-devel wget git
```

STEP 2: FreeRADIUS installation

The Kea RADIUS hook library uses the FreeRADIUS client library to conduct RADIUS communication. Unfortunately, the standard 1.1.7 release available from the project website http://freeradius.org/sub_projects/ has several serious



deficiencies; ISC engineers observed a segmentation fault during testing. Also, the base version of the library does not offer asynchronous transmissions, which are essential for effective accounting implementation. Both of these issues were addressed by ISC engineers, and the changes have been reported to the FreeRADIUS client project. Acceptance of those changes is outside of ISC's control, so until those are processed, it is strongly recommended to use the FreeRADIUS client with ISC's patches. To download and compile this version, please use the following steps:

```
$ git clone https://github.com/fxdupont/freeradius-client.git
$ cd freeradius-client/
$ git checkout iscddev
$ ./configure
$ make
$ sudo make install
```

You may pass additional parameters to the configure script, if you need to. Once installed, the FreeRADIUS client will be installed in `/usr/local`. This is the default path where Kea will be looking for it. You may install it in a different directory; if you choose to do so, make sure you add that path to the configure script when compiling Kea.

STEP 3: Install recent BOOST version

Kea requires a reasonably recent Boost version. Unfortunately, the version available in CentOS 7 is too old, so a newer Boost version is necessary. Furthermore, CentOS 7 has an old version of the g++ compiler that does not handle the latest Boost versions. Fortunately, Boost 1.65 meets both requirements; it is both recent enough for Kea and able to be compiled using the g++ 4.8 version in CentOS.

To download and compile Boost 1.65, please use the following commands:

```
$ wget -nd https://dl.bintray.com/boostorg/release/1.65.1/source/boost_1_65_1.tar.gz
$ tar zxvf boost_1_65_1.tar.gz
$ cd boost_1_65_1/
$ ./bootstrap.sh
$ ./b2 --without-python
$ sudo ./b2 install
```

Note that the b2 script may optionally take extra parameters; one of them specifies the destination path where the sources are to be compiled.

STEP 4: Compile and install Kea

Obtain the Kea sources either by downloading them from the git repository or extracting the tarball:

```
# Use one of those commands to obtain the Kea sources:

# Choice 1: get from github
$ git clone https://github.com/isc-projects/kea

# Get a tarball and extract it
$ tar zxvf kea-1.6.0-beta2.tar.gz
```

The next step is to extract the premium Kea package that contains the RADIUS repository into the Kea sources. After the tarball is extracted, the Kea sources should have a `premium/` subdirectory.

```
$ cd kea
$ tar zxvf ../kea-premium-radius-1.6.0-beta2.tar.gz
```

Once this is done, verify that the Kea sources look similar to this:

```
$ ls -l
total 952
-rw-r--r--  1 thomson  staff    6192 Apr 25 17:38 AUTHORS
-rw-r--r--  1 thomson  staff   29227 Apr 25 17:38 COPYING
-rw-r--r--  1 thomson  staff  360298 Apr 25 20:00 ChangeLog
-rw-r--r--  1 thomson  staff    645 Apr 25 17:38 INSTALL
-rw-r--r--  1 thomson  staff   5015 Apr 25 17:38 Makefile.am
```



```
-rw-r--r-- 1 thomson staff 587 Apr 25 17:38 README
-rw-r--r-- 1 thomson staff 62323 Apr 25 17:38 configure.ac
drwxr-xr-x 12 thomson staff 408 Apr 26 19:04 doc
drwxr-xr-x 7 thomson staff 238 Apr 25 17:38 examples
drwxr-xr-x 5 thomson staff 170 Apr 26 19:04 ext
drwxr-xr-x 8 thomson staff 272 Apr 26 19:04 m4macros
drwxr-xr-x 20 thomson staff 680 Apr 26 11:22 premium
drwxr-xr-x 10 thomson staff 340 Apr 26 19:04 src
drwxr-xr-x 14 thomson staff 476 Apr 26 19:04 tools
```

The makefiles must be regenerated using autoreconf.

The next step is to configure Kea, and there are several essential steps necessary here. Running autoreconf -if is necessary to compile the premium package that contains RADIUS. Also, the --with-freeradius option is necessary to tell Kea where the FreeRADIUS client sources can be found. Also, since the non-standard Boost is used, the path to it must be specified.

```
$ autoreconf -i
$ ./configure --with-freeradius=/path/to/freeradius --with-boost-include=/path/to/boost -- ←
  with-boost-lib-dir=/path/to/boost/state/lib
```

For example, assuming the FreeRADIUS client was installed in the default directory (/usr/local) and the Boost 1.65 sources were compiled in /home/thomson/devel/boost1_65_1, the configure path should look as follows:

```
./configure --with-freeradius=/usr/local \
  --with-boost-include=/home/thomson/devel/boost1_65_1 \
  --with-boost-lib-dir=/home/thomson/devel/boost1_65_1/stage/lib
```

After some checks, the configure script should print a report similar to the following:

```
Kea source configure results:
-----

Package:
  Name:          kea
  Version:       1.6.0-beta2
  Extended version: 1.6.0-beta2 (tarball)
  OS Family:    Linux

  Hooks directory: /usr/local/lib/kea/hooks
  Premium hooks:  yes
  Included Hooks:  forensic_log flex_id host_cmds subnet_cmds radius host_cache

C++ Compiler:
  CXX:           g++ --std=c++11
  CXX_VERSION:   g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-16)
  CXX_STANDARD:  201103
  DEFS:         -DHAVE_CONFIG_H
  CPPFLAGS:     -DOS_LINUX -DBOOST_ASIO_HEADER_ONLY
  CXXFLAGS:     -g -O2
  LDFLAGS:      -lpthread
  KEA_CXXFLAGS: -Wall -Wextra -Wnon-virtual-dtor -Wwrite-strings -Woverloaded-virtual - ←
  Wno-sign-compare -pthread -Wno-missing-field-initializers -fPIC

Python:
  PYTHON_VERSION: not needed (because kea-shell is disabled)

Boost:
  BOOST_VERSION: 1.65.1
  BOOST_INCLUDES: -I/home/thomson/devel/boost1_65_1
  BOOST_LIBS:    -L/home/thomson/devel/boost1_65_1/stage/lib -lboost_system
```



```
OpenSSL:
CRYPTO_VERSION:  OpenSSL 1.0.2k  26 Jan 2017
CRYPTO_CFLAGS:
CRYPTO_INCLUDES:
CRYPTO_LDFLAGS:
CRYPTO_LIBS:      -lcrypto

Botan: no

Log4cplus:
LOG4CPLUS_VERSION:  1.1.3
LOG4CPLUS_INCLUDES: -I/usr/include
LOG4CPLUS_LIBS:     -L/usr/lib -L/usr/lib64 -llog4cplus

Flex/bison:
FLEX: flex
BISON: bison -y

MySQL:
no

PostgreSQL:
no

Cassandra CQL:
no

Google Test:
no

Google Benchmark:
no

FreeRADIUS client:
FREERADIUS_INCLUDE:  -I/usr/local/include
FREERADIUS_LIB:      -L/usr/local/lib -lfreeradius-client
FREERADIUS_DICTIONARY: /usr/local/etc/radiusclient/dictionary

Developer:
Enable Debugging:    no
Google Tests:        no
Valgrind:            not found
C++ Code Coverage:  no
Logger checks:       no
Generate Documentation: no
Parser Generation:  no
Kea-shell:           no
Perfdhcp:            no
```

Please make sure that your compilation has the following:

- RADIUS listed in Included Hooks;
- FreeRADIUS client directories printed and pointing to the right directories;
- Boost version at least 1.65.1. The versions available in CentOS 7 (1.48 and 1.53) are too old.

Once your configuration is complete, compile Kea using make. If your system has more than one core, it is recommended that you use the "-j N" option to speed up the build.

```
$ make -j5
$ sudo make install
```



15.4.11.2 RADIUS Hook Configuration

The RADIUS hook is a library that has to be loaded by either DHCPv4 or DHCPv6 Kea servers. Unlike some other available hook libraries, this one takes many parameters. For example, this configuration could be used:

```
"Dhcp4": {  
  
  // Your regular DHCPv4 configuration parameters here.  
  
  "hooks-libraries": [  
    {  
      // Note that RADIUS requires host-cache for proper operation,  
      // so that library is loaded as well.  
      "library": "/usr/local/lib/kea/hooks/libdhcp_host_cache.so"  
    },  
    {  
      "library": "/usr/local/lib/kea/hooks/libdhc_radius.so",  
      "parameters": {  
  
        // Specify where FreeRADIUS dictionary could be located  
        "dictionary": "/usr/local/etc/freeradius/dictionary",  
  
        // Specify which address to use to communicate with RADIUS servers  
        "bindaddr": "*",  
  
        // more RADIUS parameters here  
      }  
    }  
  ]  
}
```

RADIUS is a complicated environment. As such, it's not really possible to provide a default configuration that would work out of the box. However, we do have one example that showcases some of the more common features. Please see `doc/examples/kea4/hooks-radius.json` in your Kea sources.

The RADIUS hook library supports the following global configuration flags, which correspond to FreeRADIUS client library options:

- **bindaddr** (default `"*"`) specifies the address to be used by the hook library in communication with RADIUS servers. The `"*"` special value tells the kernel to choose the address.
- **canonical-mac-address** (default `false`) specifies whether MAC addresses in attributes follow the canonical RADIUS format (lowercase pairs of hexadecimal digits separated by `'-'`).
- **client-id-pop0** (default `false`), used with `flex-id`, removes the leading zero (or pair of zeroes in DHCPv6) type in `client-id` (aka `duid` in DHCPv6). Implied by `client-id-printable`.
- **client-id-printable** (default `false`) checks whether the `client-id/duid` content is printable and uses it as is instead of in hexadecimal. Implies `client-id-pop0` and `extract-duid` as 0 and 255 are not printable.
- **deadtime** (default 0) is a mechanism to try unresponsive servers after responsive servers. Its value specifies the number of seconds after which a server is considered not to have answered, so 0 disables the mechanism. As the asynchronous communication does not use locks or atomics, it is recommended that you do not use this feature when running in this mode.
- **dictionary** (default set by configure at build time) is the attribute and value dictionary. Note that it is a critical parameter.
- **extract-duid** (default `true`) extracts the embedded `duid` from an RFC 4361-compliant DHCPv4 `client-id`. Implied by `client-id-printable`.
- **identifier-type4** (default `client-id`) specifies the identifier type to build the `User-Name` attribute. It should be the same as the host identifier, and when the `flex-id` hook library is used the `replace-client-id` must be set to `true`; `client-id` will be used with `client-id-pop0`.



- **identifier-type6** (default duid) specifies the identifier type to build the User-Name attribute. It should be the same as the host identifier, and when the flex-id hook library is used the `replace-client-id` must be set to true; duid will be used with `client-id-pop0`.
- **realm** (default "") is the default realm.
- **reselect-subnet-address** (default false) uses the Kea reserved address/RADIUS Framed-IP-Address or Framed-IPv6-Address to reselect subnets where the address is not in the subnet range.
- **reselect-subnet-pool** (default false) uses the Kea client-class/RADIUS Frame-Pool to reselect subnets where no available pool can be found.
- **retries** (default 3) is the number of retries before trying the next server. Note that it is not supported for asynchronous communication.
- **session-history** (default "") is the name of the file providing persistent storage for accounting session history.
- **timeout** (default 10) is the number of seconds during which a response is awaited.

When **reselect-subnet-pool** or **reselect-subnet-address** is set to true at the reception of RADIUS Access-Accept, the selected subnet is checked against the client-class name or the reserved address; if it does not match, another subnet is selected among matching subnets.

Two services are supported:

- **access** - the authentication service
- **accounting** - the accounting service

Configuration of services is divided into two parts:

- servers that define RADIUS servers the library is expected to contact. Each server may have the following items specified:
 - **name**, which specifies the IP address of the server (it is possible to use a name which will be resolved, but it is not recommended).
 - **port** (default RADIUS authentication or accounting service), which specifies the UDP port of the server. Note that the FreeRADIUS client library by default uses ports 1812 (authorization) and 1813 (accounting). Some server implementations use 1645 (authorization) and 1646 (accounting). You may use the "port" parameter to adjust as needed.
 - **secret**, which authenticates messages.

There may be up to eight servers. Note that when no server is specified, the service is disabled.

- attributes which define additional information that the Kea server will send to a RADIUS server. The parameter must be identified either by a name or type. Its value can be specified in one of three possible ways: `data` (which defines a plain text value), `raw` (which defines the value in hex), or `expr` (which defines an expression, which will be evaluated for each incoming packet independently).
 - **name** of the attribute.
 - **type** of the attribute. Either the type or the name must be provided, and the attribute must be defined in the dictionary.
 - **data** is the first of three ways to specify the attribute content. The data entry is parsed by the FreeRADIUS library, so values defined in the dictionary of the attribute may be used.
 - **raw** is the second of three ways to specify the attribute content; it specifies the content in hexadecimal. Note that it does not work with integer-content attributes (date, integer, and IPv4 address); a string-content attribute (string, IPv6 address, and IPv6 prefix) is required.
 - **expr** is the last way to specify the attribute content. It specifies an evaluation expression which must return a not-empty string when evaluated with the DHCP query packet. Currently this is restricted to the access service.

For example, to specify a single access server available on localhost that uses "xyz123" as a secret, and tell Kea to send three additional attributes (Password, Connect-Info, and Configuration-Token), the following snippet could be used:



```
"parameters": {  
    // Other RADIUS parameters here  
  
    "access": {  
  
        // This starts the list of access servers  
        "servers": [  
            {  
                // These are parameters for the first (and only) access server  
                "name": "127.0.0.1",  
                "port": 1812,  
                "secret": "xyz123"  
            }  
            // Additional access servers could be specified here  
        ],  
  
        // This defines a list of additional attributes Kea will send to each  
        // access server in Access-Request.  
        "attributes": [  
            {  
                // This attribute is identified by name (must be present in the  
                // dictionary) and has static value (i.e. the same value will be  
                // sent to every server for every packet)  
                "name": "Password",  
                "data": "mysecretpassword"  
            },  
            {  
                // It's also possible to specify an attribute using its type,  
                // rather than a name. 77 is Connect-Info. The value is specified  
                // using hex. Again, this is a static value. It will be sent the  
                // same for every packet and to every server.  
                "type": 77,  
                "raw": "65666a6a71"  
            },  
            {  
                // This example shows how an expression can be used to send dynamic  
                // value. The expression (see Section 13) may take any value from  
                // the incoming packet or even its metadata (e.g. the interface  
                // it was received over from)  
                "name": "Configuration-Token",  
                "expr": "hexstring(pkt4.mac, ':')"  
            }  
        ] // End of attributes  
    } // End of access  
  
    // Accounting parameters.  
    "accounting": {  
        // This starts the list of accounting servers  
        "servers": [  
            {  
                // These are parameters for the first (and only) accounting server  
                "name": "127.0.0.1",  
                "port": 1813,  
                "secret": "sekret"  
            }  
            // Additional accounting servers could be specified here  
        ]  
    }  
}
```



For the RADIUS hook library to operate properly in DHCPv4, it is necessary to also load the Host Cache hook library. The reason for this is somewhat complex. In a typical deployment, the DHCP clients send their packets via DHCP relay which inserts certain Relay Agent Information options, such as circuit-id or remote-id. The values of those options are then used by the Kea DHCP server to formulate the necessary attributes in the Access-Request message sent to the RADIUS server. However, once the DHCP client gets its address, it then renews by sending packets directly to the DHCP server. As a result, the relays are not able to insert their RAI options and the DHCP server can't send the Access-Request queries to the RADIUS server by using just the information from incoming packets. Kea needs to keep the information received during the initial Discover/Offer exchanges and use it again later when sending accounting messages.

This mechanism is implemented based on user context in host reservations. (See Section 15.5 for details about user context). The host cache mechanism allows the information retrieved by RADIUS to be stored and later used for sending accounting and access queries to the RADIUS server. In other words, the host-cache mechanism is mandatory, unless you do not want RADIUS communication for messages other than Discover and the first Request from each client.

15.4.12 host_cache: Caching Host Reservations

Some database backends, such as RADIUS, are considered slow and may take a long time to respond. Since Kea in general is synchronous, the backend performance directly affects the DHCP performance. To minimize the impact and improve performance, the Host Cache library provides a way to cache information from the database locally. This includes negative caching, i.e. the ability to remember that there is no client information in the database.

Note

This library may only be loaded by the **kea-dhcp4** or the **kea-dhcp6** process.

In principle, this hook library can be used with any backend that may introduce performance degradation (MySQL, PostgreSQL, Cassandra, RADIUS). Host Cache must be loaded for the RADIUS accounting mechanism to work.

The Host Cache hook library is currently very simple. It takes only one optional parameter ("maximum") that defines the maximum number of hosts to be cached. If not specified, the default value of 0 is used, which means there is no limit. The hook library can be loaded the same way as any other hook library; for example, this configuration could be used:

```
"Dhcp4": {  
  
  // Your regular DHCPv4 configuration parameters here.  
  
  "hooks-libraries": [  
    {  
      "library": "/usr/local/lib/kea/hooks/libdhc_host_cache.so",  
      "parameters": {  
  
        // Tells Kea to never cache more than 1000 hosts.  
        "maximum": 1000  
  
      }  
    }  
  ]  
}
```

Once loaded, the Host Cache hook library provides a number of new commands which can be used either over the control channel (see Section 17.2) or the REST API (see Section 7.1). An example REST API client is described in Section 19.1. The following sections describe the commands available.

15.4.12.1 cache-flush Command

This command allows removal of a specified number of cached host entries. It takes one parameter, which defines the number of hosts to be removed. An example usage looks as follows:



```
{
  "command": "cache-flush",
  "arguments": 1000
}
```

This command will remove 1000 hosts. If you want to delete all cached hosts, please use `cache-clear` instead. The hosts are stored in FIFO order, so the oldest entries are always removed.

15.4.12.2 `cache-clear` Command

This command allows removal of all cached host entries. An example usage looks as follows:

```
{
  "command": "cache-clear"
}
```

This command will remove all hosts. If you want to delete only a certain number of cached hosts, please use `cache-flush` instead.

15.4.12.3 `cache-size` Command

This command returns the number of host entries. An example usage looks as follows:

```
{
  "command": "cache-size"
}
```

15.4.12.4 `cache-write` Command

In general, the cache content is considered a runtime state and the server can be shut down or restarted as usual; the cache will then be repopulated after restart. However, there are some cases when it is useful to store the contents of the cache. One such case is RADIUS, where the cached hosts also retain additional cached RADIUS attributes; there is no easy way to obtain this information again, because renewing clients send their packet to the DHCP server directly. Another use case is when you want to restart the server and for performance reasons you want it to start with a hot (populated) cache.

This command allows writing the contents of the in-memory cache to a file on disk. It takes one parameter, which defines the filename. An example usage looks as follows:

```
{
  "command": "cache-write",
  "arguments": "/tmp/kea-host-cache.json"
}
```

This causes the contents to be stored in the `/tmp/kea-host-cache.json` file. That file can then be loaded with the `cache-load` command or processed by any other tool that is able to understand JSON format.

15.4.12.5 `cache-load` Command

See the previous section for a discussion of use cases where it may be useful to write and load contents of the host cache to disk.

This command allows the contents of a file on disk to be loaded into an in-memory cache. It takes one parameter, which defines the filename. An example usage looks as follows:

```
{
  "command": "cache-load",
  "arguments": "/tmp/kea-host-cache.json"
}
```

This command will store the contents to the `/tmp/kea-host-cache.json` file. That file can then be loaded with the `cache-load` command or processed by any other tool that is able to understand JSON format.



15.4.12.6 cache-get Command

This command is similar to cache-write, but instead of writing the cache contents to disk, it returns the contents to whoever sent the command.

This command allows the contents of a file on disk to be loaded into an in-memory cache. It takes one parameter, which defines the filename. An example usage looks as follows:

```
{
  "command": "cache-get"
}
```

This command will return all the cached hosts. Note that the response may be large.

15.4.12.7 cache-get-by-id Command

This command is similar to cache-get, but instead of returning the whole content it returns only the entries matching the given identifier.

It takes one parameter, which defines the identifier of wanted cached host reservations. An example usage looks as follows:

```
{
  "command": "cache-get-by-id",
  "arguments": {
    "hw-address": "01:02:03:04:05:06"
  }
}
```

This command will return all the cached hosts with the given hardware address.

15.4.12.8 cache-insert Command

This command may be used to manually insert a host into the cache; there are very few use cases when this command could be useful. This command expects its arguments to follow the usual syntax for specifying host reservations (see Section 8.3 or Section 9.3), with one difference: the subnet-id value must be specified explicitly.

An example command that will insert an IPv4 host into the host cache looks as follows:

```
{
  "command": "cache-insert",
  "arguments": {
    "hw-address": "01:02:03:04:05:06",
    "subnet-id4": 4,
    "subnet-id6": 0,
    "ip-address": "192.0.2.100",
    "hostname": "somehost.example.org",
    "client-classes4": [ ],
    "client-classes6": [ ],
    "option-data4": [ ],
    "option-data6": [ ],
    "next-server": "192.0.0.2",
    "server-hostname": "server-hostname.example.org",
    "boot-file-name": "bootfile.efi",
    "host-id": 0
  }
}
```

An example command that will insert an IPv6 host into the host cache looks as follows:



```
{
  "command": "cache-insert",
  "arguments": {
    "hw-address": "01:02:03:04:05:06",
    "subnet-id4": 0,
    "subnet-id6": 6,
    "ip-addresses": [ "2001:db8::cafe:babe" ],
    "prefixes": [ "2001:db8:dead:beef::/64" ],
    "hostname": "",
    "client-classes4": [ ],
    "client-classes6": [ ],
    "option-data4": [ ],
    "option-data6": [ ],
    "next-server": "0.0.0.0",
    "server-hostname": "",
    "boot-file-name": "",
    "host-id": 0
  }
}
```

15.4.12.9 cache-remove Command

Sometimes it is useful to remove a single entry from the host cache. A good use case is a situation where the device is up, Kea has already provided configuration, and the host entry is in cache. As a result of administrative action (e.g. customer hasn't paid their bills or has perhaps been upgraded to better service), the information in the backend (e.g. MySQL or RADIUS) is being updated. However, since cache is in use, Kea does not notice the change as the cached values are used. The cache-remove command can solve this problem by removing a cached entry after administrative changes.

The cache-remove command works similarly to the reservation-get command. It allows querying by two parameters: either subnet-id4 or subnet-id6; or ip-address (may be an IPv4 or IPv6 address), hw-address (specifies hardware/MAC address), duid, circuit-id, client-id, or flex-id.

An example command to remove an IPv4 host with reserved address 192.0.2.1 from a subnet with a subnet-id 123 looks as follows:

```
{
  "command": "cache-remove",
  "arguments": {
    "ip-address": "192.0.2.1",
    "subnet-id": 123
  }
}
```

Another example that removes an IPv6 host identifier by DUID and specific subnet-id is:

```
{
  "command": "cache-remove",
  "arguments": {
    "duid": "00:01:ab:cd:f0:a1:c2:d3:e4",
    "subnet-id": 123
  }
}
```

15.5 User contexts

Hook libraries can have their own configuration parameters. That is convenient if the parameter applies to the whole library. However, sometimes it is very useful if certain configuration entities are extended with additional configuration data. This is



where the concept of user contexts comes in. A sysadmin can define an arbitrary set of data and attach it to Kea structures, as long as the data is specified as JSON map. In particular, it is possible to define fields that are integers, strings, boolean, lists and maps. It is possible to define nested structures of arbitrary complexity. Kea does not use that data on its own, simply stores and makes it available for the hook libraries.

Another use case for user contexts may be storing comments and other information that will be retained by Kea. Regular comments are discarded when configuration is loaded, but user contexts are retained. This is useful if you want your comments to survive config-set, config-get operations for example.

If user context is supported in a given context, the parser translates "comment" entries into user context with a "comment" entry. The pretty print of a configuration does the opposite operation and puts "comment" entries at the beginning of maps as it seems to be the common usage.

As of Kea 1.3, the structures that allow user contexts are pools of all types (addresses and prefixes) and subnets. Kea 1.4 extended user context support to the global scope, interfaces config, shared networks, subnets, client classes, option datas and definitions, host reservations, control socket, dhcp ddns, loggers and server id. These are supported in both DHCPv4 and DHCPv6 at the exception of server id which is DHCPv6 only.



Chapter 16

Statistics

16.1 Statistics Overview

Both Kea DHCP servers support statistics gathering. A working DHCP server encounters various events that can cause certain statistics to be collected. For example, a DHCPv4 server may receive a packet (pkt4-received statistic increases by one) that after parsing is identified as a DHCPDISCOVER (pkt4-discover-received). The server processes it and decides to send a DHCPOFFER representing its answer (pkt4-offer-sent and pkt4-sent statistics increase by one). Such events happen frequently, so it is not uncommon for the statistics to have values in the high thousands. They can serve as an easy and powerful tool for observing a server's and a network's health. For example, if the pkt4-received statistic stops growing, it means that the clients' packets are not reaching the server.

There are four types of statistics:

- *integer* - this is the most common type. It is implemented as a 64-bit integer (int64_t in C++), so it can hold any value between -2^{63} to $2^{63} - 1$.
- *floating point* - this type is intended to store floating-point precision. It is implemented as a C++ double type.
- *duration* - this type is intended for recording time periods. It uses the `boost::posix_time::time_duration` type, which stores hours, minutes, seconds, and microseconds.
- *string* - this type is intended for recording statistics in textual form. It uses the C++ `std::string` type.

During normal operation, the DHCPv4 and DHCPv6 servers gather statistics. For a list of DHCPv4 and DHCPv6 statistics, see Section 8.8 and Section 9.13, respectively.

To extract data from the statistics module, the control channel can be used. See Chapter 17 for details. It is possible to retrieve a single statistic or all statistics, reset statistics (i.e. set to neutral value, typically zero), or even remove completely a single statistic or all statistics. See section Section 16.3 for a list of statistics-oriented commands.

16.2 Statistics Lifecycle

It is useful to understand how the Statistics Manager module works. When the server starts operation, the manager is empty and contains no statistics. When **statistic-get-all** is executed, an empty list is returned. Once the server performs an operation that causes a statistic to change, the related statistic will be created. In general, once a statistic is recorded even once, it is kept in the manager until explicitly removed, by **statistic-remove** or **statistic-remove-all** being called, or when the server is shut down. Per-subnet statistics are explicitly removed when reconfiguration takes place.

Statistics are considered runtime properties, so they are not retained after server restart.

Removing a statistic that is updated frequently makes little sense, as it will be re-added when the server code next records that statistic. The **statistic-remove** and **statistic-remove-all** commands are intended to remove statistics that are not expected to be



observed in the near future. For example, a misconfigured device in a network may cause clients to report duplicate addresses, so the server will report increasing values of `pkt4-decline-received`. Once the problem is found and the device is removed, the system administrator may want to remove the `pkt4-decline-received` statistic, so it won't be reported anymore. If a duplicate address is detected ever again, the server will add this statistic back.

16.3 Commands for Manipulating Statistics

There are several commands defined that can be used for accessing (`-get`), resetting to zero or neutral value (`-reset`), or even removing a statistic completely (`-remove`). The difference between `reset` and `remove` is somewhat subtle. The `reset` command sets the value of the statistic to zero or a neutral value, so after this operation, the statistic will have a value of 0 (integer), 0.0 (float), 0h0m0s0us (duration), or "" (string). When requested, a statistic with the values mentioned will be returned. **Remove** removes a statistic completely, so the statistic will not be reported anymore. Please note that the server code may add it back if there is a reason to record it.

Note

The following sections describe commands that can be sent to the server; the examples are not fragments of a configuration file. For more information on sending commands to Kea, see Chapter 17.

16.3.1 `statistic-get` Command

The `statistic-get` command retrieves a single statistic. It takes a single-string parameter called **name**, which specifies the statistic name. An example command may look like this:

```
{
  "command": "statistic-get",
  "arguments": {
    "name": "pkt4-received"
  }
}
```

The server returns details of the requested statistic, with a result of 0 indicating success and the specified statistic as the value of the "arguments" parameter. If the requested statistic is not found, the response will contain an empty map, i.e. only { } as an argument, but the status code will still indicate success (0).

16.3.2 `statistic-reset` Command

The `statistic-reset` command sets the specified statistic to its neutral value: 0 for integer, 0.0 for float, 0h0m0s0us for time duration, and "" for string type. It takes a single-string parameter called **name**, which specifies the statistic name. An example command may look like this:

```
{
  "command": "statistic-reset",
  "arguments": {
    "name": "pkt4-received"
  }
}
```

If the specific statistic is found and the reset is successful, the server responds with a status of 0, indicating success, and an empty parameters field. If an error is encountered (e.g. the requested statistic was not found), the server returns a status code of 1 (error) and the text field contains the error description.



16.3.3 statistic-remove Command

The *statistic-remove* command attempts to delete a single statistic. It takes a single-string parameter called **name**, which specifies the statistic name. An example command may look like this:

```
{
  "command": "statistic-remove",
  "arguments": {
    "name": "pkt4-received"
  }
}
```

If the specific statistic is found and its removal is successful, the server responds with a status of 0, indicating success, and an empty parameters field. If an error is encountered (e.g. the requested statistic was not found), the server returns a status code of 1 (error) and the text field contains the error description.

16.3.4 statistic-get-all Command

The *statistic-get-all* command retrieves all statistics recorded. An example command may look like this:

```
{
  "command": "statistic-get-all",
  "arguments": { }
}
```

The server responds with details of all recorded statistics, with a result set to 0 to indicate that it iterated over all statistics (even when the total number of statistics is zero).

16.3.5 statistic-reset-all Command

The *statistic-reset* command sets all statistics to their neutral values: 0 for integer, 0.0 for float, 0h0m0s0us for time duration, and "" for string type. An example command may look like this:

```
{
  "command": "statistic-reset-all",
  "arguments": { }
}
```

If the operation is successful, the server responds with a status of 0, indicating success, and an empty parameters field. If an error is encountered, the server returns a status code of 1 (error) and the text field contains the error description.

16.3.6 statistic-remove-all Command

The *statistic-remove-all* command attempts to delete all statistics. An example command may look like this:

```
{
  "command": "statistic-remove-all",
  "arguments": { }
}
```

If the removal of all statistics is successful, the server responds with a status of 0, indicating success, and an empty parameters field. If an error is encountered, the server returns a status code of 1 (error) and the text field will contain the error description.



Chapter 17

Management API

A classic approach to daemon configuration assumes that the server's configuration is stored in configuration files and, when the configuration is changed, the daemon is restarted. This approach has the significant disadvantage of introducing periods of downtime when client traffic is not handled. Another risk is that if the new configuration is invalid for any reason, the server may refuse to start, which will further extend the downtime period until the issue is resolved.

To avoid such problems, the DHCPv4, DHCPv6 and D2 servers in Kea include support for a mechanism that allows online reconfiguration without requiring server shutdown. Both servers can be instructed to open control sockets, which is a communications channel. The server is able to receive commands on that channel, act on them, and report back status.

The DHCPv4, DHCPv6 and D2 servers receive commands over the UNIX domain sockets. The details how to configure these sockets, see Section 8.9 and Section 9.14. While it is possible to control the servers directly using unix domain sockets it requires that the controlling client be running on the same machine as the server. SSH is usually used to connect remotely to the controlled machine.

Network administrators usually prefer using some form of a RESTful API to control the servers, rather than using UNIX domain sockets directly. Therefore, Kea includes a component called Control Agent (or CA), which exposes a RESTful API to the controlling clients and can forward commands to the respective Kea services over the UNIX domain sockets. The CA configuration has been described in Section 7.2.

The HTTP requests received by the CA contain the control commands encapsulated within HTTP requests. Simply speaking, the CA is responsible for stripping the HTTP layer from the received commands and forwarding the commands in a JSON format over the UNIX domain sockets to the respective services. Because the CA receives commands for all services, it requires additional "forwarding" information to be included in the client's messages. This forwarding information is carried within the **service** parameter of the received command. If the **service** parameter is not included, or if the parameter is a blank list, the CA will assume that the control command is targeted at the CA itself and will try to handle it on its own.

Control connections over both HTTP and UNIX domain sockets are guarded with timeouts. The default timeout value is set to 10s and is not configurable.

17.1 Data Syntax

Communication over the control channel is conducted using JSON structures. If configured, Kea will open a socket and listen for incoming connections. A process connecting to this socket is expected to send JSON commands structured as follows:

```
{
  "command": "foo",
  "service": [ "dhcp4" ]
  "arguments": {
    "param1": "value1",
    "param2": "value2",
    ...
  }
}
```



The same command sent over the RESTful interface to the CA will have the following structure:

```
POST / HTTP/1.1\r\n
Content-Type: application/json\r\n
Content-Length: 147\r\n\r\n
{\n
  "command": "foo",\n
  "service": [ "dhcp4" ]\n
  "arguments": {\n
    "param1": "value1",\n
    "param2": "value2",\n
    ...\n
  }\n
}
```

command is the name of the command to execute and is mandatory. **arguments** is a map of parameters required to carry out the given command. The exact content and format of the map is command-specific.

service is a list of the servers at which the control command is targeted. In the example above, the control command is targeted at the DHCPv4 server. In most cases, the CA will simply forward this command to the DHCPv4 server for processing via a UNIX domain socket. Sometimes, the command including a service value may also be processed by the CA, if the CA is running a hooks library which handles such a command for the given server. As an example, the hooks library loaded by the CA may perform some operations on the database, such as adding host reservations, modifying leases, etc. An advantage of performing DHCPv4-specific administrative operations in the CA, rather than forwarding it to the DHCPv4 server, is the ability to perform these operations without disrupting the DHCPv4 service, since the DHCPv4 server doesn't have to stop processing DHCP messages to apply changes to the database. Nevertheless, these situations are rather rare and, in most cases, when the **service** parameter contains a name of the service the commands are simply forwarded by the CA. The forwarded command includes the **service** parameter but this parameter is ignored by the receiving server. This parameter is only meaningful to the CA.

If the command received by the CA does not include a **service** parameter or this list is empty, the CA simply processes this message on its own. For example, a **config-get** command which includes no service parameter returns the Control Agent's own configuration. The **config-get** with a service value "dhcp4" is forwarded to the DHCPv4 server and returns the DHCPv4 server's configuration.

The following list shows the mapping of the values carried within the **service** parameter to the servers to which the commands are forwarded:

- **dhcp4** - the command is forwarded to the **kea-dhcp4** server.
- **dhcp6** - the command is forwarded to the **kea-dhcp6** server.
- **d2** - the command is forwarded to the **kea-d2** server.

The server processing the incoming command will send a response of the form:

```
{\n  "result": 0|1|2|3,\n  "text": "textual description",\n  "arguments": {\n    "argument1": "value1",\n    "argument2": "value2",\n    ...\n  }\n}
```

result indicates the outcome of the command. A value of 0 means success, while any non-zero value designates an error or a failure to complete the requested action. Currently 1 indicates a generic error, 2 means that a command is not supported, and 3 means that the requested operation was completed, but the requested object was not found. For example, a well-formed command that requests a subnet that exists in a server's configuration returns the result 0. If the server encounters an error condition, it returns 1. If the command asks for the IPv6 subnet, but was sent to a DHCPv4 server, it returns 2. If the query asks for a subnet-id and there is no subnet with such an id, the result is 3.



The **text** field typically appears when the result is non-zero and contains a description of the error encountered, but it often also appears for successful outcomes. The exact text is command-specific, but in general uses plain English to describe the outcome of the command. **arguments** is a map of additional data values returned by the server which are specific to the command issued. The map may be present, but that depends on the specific command.

Note

When sending commands via Control Agent, it is possible to specify multiple services at which the command is targeted. CA forwards this command to each service individually. Thus, the CA response to the controlling client contains an array of individual responses.

17.2 Using the Control Channel

The easiest way to start interacting with the control API is to use common UNIX/Linux tools such as **socat** and **curl**.

In order to control the given Kea service via UNIX domain socket, use **socat** in interactive mode as follows:

```
$ socat UNIX:/path/to/the/kea/socket -
```

or in batch mode, include the "ignoreeof" option as shown below to ensure socat waits long enough for the server to respond:

```
$ echo "{ some command...}" | socat UNIX:/path/to/the/kea/socket -,ignoreeof
```

where **/path/to/the/kea/socket** is the path specified in the **Dhcp4/control-socket/socket-name** parameter in the Kea configuration file. Text passed to **socat** is sent to Kea and the responses received from Kea are printed to standard output. This approach communicates with the specific server directly and bypasses the Control Agent.

It is also easy to open a UNIX socket programmatically. An example of a simple client written in C is available in the Kea Developer's Guide, in the Control Channel Overview chapter, in the Using Control Channel section.

To use Kea's RESTful API with **curl**, you may use the following:

```
$ curl -X POST -H "Content-Type: application/json" -d '{ "command": "config-get", "service ←  
": [ "dhcp4" ] }' http://ca.example.org:8000/
```

This assumes that the Control Agent is running on host `ca.example.org` and is running the RESTful service on port 8000.

17.3 Commands Supported by Both the DHCPv4 and DHCPv6 Servers

17.3.1 build-report

The *build-report* command returns on the control channel what the command line **-W** argument displays, i.e. the embedded content of the `config.report` file. This command does not take any parameters.

```
{  
  "command": "build-report"  
}
```

17.3.2 config-get

The *config-get* command retrieves the current configuration used by the server. This command does not take any parameters. The configuration returned is roughly equal to the configuration that was loaded using the **-c** command line option during server start-up or later set using `config-set` command. However, there may be certain differences, as comments are not retained. If the original configuration used file inclusion, the returned configuration will include all parameters from all the included files.



Note that the returned configuration is not redacted, i.e. it will contain database passwords in plain text if those were specified in the original configuration. Care should be taken not to expose the command channel to unprivileged users.

An example command invocation looks like this:

```
{
  "command": "config-get"
}
```

17.3.3 config-reload

The *config-reload* command instructs Kea to load again the configuration file that was used previously. This operation is useful if the configuration file has been changed by some external source; for example, a sysadmin can tweak the configuration file and use this command to force Kea pick up the changes.

Caution should be taken when mixing this with *config-set* commands. Kea remembers the location of the configuration file it was started with, and this configuration can be significantly changed using *config-set* command. When *config-reload* is issued after *config-set*, Kea will attempt to reload its original configuration from the file, possibly losing all changes introduced using *config-set* or other commands.

config-reload does not take any parameters. An example command invocation looks like this:

```
{
  "command": "config-reload"
}
```

17.3.4 config-test

The *config-test* command instructs the server to check whether the new configuration supplied in the command's arguments can be loaded. The supplied configuration is expected to be the full configuration for the target server, along with an optional Logger configuration. As for the *-t* command, some sanity checks are not performed so it is possible a configuration which successfully passes this command will still fail in the *config-set* command or at launch time. The structure of the command is as follows:

```
{
  "command": "config-test",
  "arguments": {
    "<server>": {
    }
  }
}
```

where *<server>* is the configuration element name for a given server such as "Dhcp4" or "Dhcp6". For example:

```
{
  "command": "config-test",
  "arguments": {
    "Dhcp6": {
      :
    }
  }
}
```

The server's response will contain a numeric code, "result" (0 for success, non-zero on failure), and a string, "text", describing the outcome:

```
{"result": 0, "text": "Configuration seems sane..."}
or
{"result": 1, "text": "unsupported parameter: BOGUS (<string>:16:26)" }
```



17.3.5 config-write

The *config-write* command instructs the Kea server to write its current configuration to a file on disk. It takes one optional argument called *filename* that specifies the name of the file to write the configuration to. If not specified, the name used when starting Kea (passed as a *-c* argument) will be used. If a relative path is specified, Kea will write its files only in the directory it is running.

An example command invocation looks like this:

```
{
  "command": "config-write",
  "arguments": {
    "filename": "config-modified-2017-03-15.json"
  }
}
```

17.3.6 leases-reclaim

The *leases-reclaim* command instructs the server to reclaim all expired leases immediately. The command has the following JSON syntax:

```
{
  "command": "leases-reclaim",
  "arguments": {
    "remove": true
  }
}
```

The *remove* boolean parameter is mandatory and indicates whether the reclaimed leases should be removed from the lease database (if true), or left in the *expired-reclaimed* state (if false). The latter facilitates lease affinity, i.e. the ability to re-assign an expired lease to the same client which used this lease before. See Section 10.4 for the details. Also, see Section 10.1 for general information about the processing of expired leases (leases reclamation).

17.3.7 libreload

The *libreload* command first unloads and then loads all currently loaded hook libraries. This is primarily intended to allow one or more hook libraries to be replaced with newer versions without requiring Kea servers to be reconfigured or restarted. Note that the hook libraries are passed the same parameter values (if any) that were passed when they originally loaded.

```
{
  "command": "libreload",
  "arguments": { }
}
```

The server will respond with a result of either 0, indicating success, or 1, indicating failure.

17.3.8 list-commands

The *list-commands* command retrieves a list of all commands supported by the server. It does not take any arguments. An example command may look like this:

```
{
  "command": "list-commands",
  "arguments": { }
}
```

The server responds with a list of all supported commands. The arguments element is a list of strings, each of which conveys one supported command.



17.3.9 config-set

The *config-set* command instructs the server to replace its current configuration with the new configuration supplied in the command's arguments. The supplied configuration is expected to be the full configuration for the target server, along with an optional Logger configuration. While optional, the Logger configuration is highly recommended, as without it the server will revert to its default logging configuration. The structure of the command is as follows:

```
{
  "command": "config-set",
  "arguments": {
    "<server>": {
    }
  }
}
```

where *<server>* is the configuration element name for a given server such as "Dhcp4" or "Dhcp6". For example:

```
{
  "command": "config-set",
  "arguments": {
    "Dhcp6": {
      :
    }
  }
}
```

If the new configuration proves to be invalid, the server retains its current configuration. Please note that the new configuration is retained in memory only; if the server is restarted or a configuration reload is triggered via a signal, the server uses the configuration stored in its configuration file. The server's response contains a numeric code, "result" (0 for success, non-zero on failure), and a string, "text", describing the outcome:

```
{"result": 0, "text": "Configuration successful." }

or

{"result": 1, "text": "unsupported parameter: BOGUS (<string>:16:26)" }
```

17.3.10 shutdown

The *shutdown* command instructs the server to initiate its shutdown procedure. It is the equivalent of sending a SIGTERM signal to the process. This command does not take any arguments. An example command may look like this:

```
{
  "command": "shutdown"
}
```

The server responds with a confirmation that the shutdown procedure has been initiated.

17.3.11 dhcp-disable

The *dhcp-disable* command globally disables the DHCP service. The server continues to operate, but it drops all received DHCP messages. This command is useful when the server's maintenance requires that the server temporarily stop allocating new leases and renew existing leases. It is also useful in failover-like configurations during a synchronization of the lease databases at startup, or recovery after a failure. The optional parameter *max-period* specifies the time in seconds after which the DHCP service should be automatically re-enabled, if the *dhcp-enable* command is not sent before this time elapses.



```
{
  "command": "dhcp-disable",
  "arguments": {
    "max-period": 20
  }
}
```

17.3.12 dhcp-enable

The *dhcp-enable* command globally enables the DHCP service.

```
{
  "command": "dhcp-enable"
}
```

17.3.13 version-get

The *version-get* command returns extended information about the Kea version. It is the same information available via the **-V** command-line argument. This command does not take any parameters.

```
{
  "command": "version-get"
}
```

17.4 Commands Supported by D2 Server

The D2 server supports only a subset of DHCPv4 / DHCPv6 server commands:

- build-report
- config-get
- config-reload
- config-set
- config-test
- config-write
- list-commands
- shutdown
- version-get

17.5 Commands Supported by Control Agent

The following commands listed in Section 17.3 are also supported by the Control Agent, i.e. when the **service** parameter is blank, the commands are handled by the CA and they relate to the CA process itself:

- build-report
- config-get



- `config-reload`
 - `config-set`
 - `config-test`
 - `config-write`
 - `list-commands`
 - `shutdown`
 - `version-get`
-



Chapter 18

Logging

18.1 Logging Configuration

During its operation Kea may produce many messages. They differ in severity (some are more important than others) and source (different components (e.g. hooks) produce different messages). It is useful to understand which log messages are critical and which are not, and configure your logging appropriately. For example, debug-level messages can be safely ignored in a typical deployment. They are, however, very useful when debugging a problem.

The logging system in Kea is configured through the loggers entry in the server section of your configuration file. In previous Kea releases this entry was in an independent Logging section, this is still supported for backward compatibility.

18.1.1 Loggers

Within Kea, a message is logged through an entity called a "logger." Different components log messages through different loggers, and each logger can be configured independently of the others. Some components, in particular the DHCP server processes, may use multiple loggers to log messages pertaining to different logical functions of the component. For example, the DHCPv4 server uses one logger for messages about packet reception and transmission, another logger for messages related to lease allocation, and so on. Some of the libraries used by the Kea server, e.g. libdhcpdrv, use their own loggers.

Users implementing hooks libraries (code attached to the server at runtime) are responsible for creating the loggers used by those libraries. Such loggers should have unique names, different from the logger names used by Kea. In this way the messages produced by the hooks library can be distinguished from messages issued by the core Kea code. Unique names also allow the loggers to be configured independently of loggers used by Kea. Whenever it makes sense, a hooks library can use multiple loggers to log messages pertaining to different logical parts of the library.

In the server section of a configuration file you can specify the configuration for zero or more loggers (including loggers used by the proprietary hooks libraries). If there are no loggers specified, the code will use default values; these cause Kea to log messages of INFO severity or greater to standard output. There is a small time window after Kea has been started but before it has read its configuration; logging in this short period can be controlled using environment variables. For details, see Section [18.1.3](#).

The three main elements of a logger configuration are: **name** (the component that is generating the messages), **severity** (what to log), and **output_commands** (where to log). There is also a **debuglevel** element, which is only relevant if debug-level logging has been selected.

18.1.1.1 name (string)

Each logger in the system has a name: that of the component binary file using it to log messages. For instance, if you want to configure logging for the DHCPv4 server, you add an entry for a logger named "kea-dhcp4". This configuration will then be used by the loggers in the DHCPv4 server, and all the libraries used by it (unless a library defines its own logger and there is a specific logger configuration that applies to that logger).



When tracking down an issue with the server's operation, use of DEBUG logging is required to obtain the verbose output needed for problem diagnosis. However, the high verbosity is likely to overwhelm the logging system in cases where the server is processing high-volume traffic. To mitigate this problem, Kea can use multiple loggers, for different functional parts of the server, that can each be configured independently. If the user is reasonably confident that a problem originates in a specific function of the server, or that the problem is related to a specific type of operation, they may enable high verbosity only for the relevant logger, thereby limiting the debug messages to the required minimum.

The loggers are associated with a particular library or binary of Kea. However, each library or binary may (and usually does) include multiple loggers. For example, the DHCPv4 server binary contains separate loggers for packet parsing, for dropped packets, for callouts, etc.

The loggers form a hierarchy. For each program in Kea, there is a "root" logger, named after the program (e.g. the root logger for kea-dhcp (the DHCPv4 server) is named kea-dhcp4. All other loggers are children of this logger and are named accordingly, e.g. the allocation engine in the DHCPv4 server logs messages using a logger called kea-dhcp4.alloc-engine.

This relationship is important, as each child logger derives its default configuration from its parent root logger. In the typical case, the root logger configuration is the only logging configuration specified in the configuration file and so applies to all loggers. If an entry is made for a given logger, any attributes specified override those of the root logger, whereas any not specified are inherited from it.

To illustrate this, suppose you are using the DHCPv4 server with the root logger "kea-dhcp4" logging at the INFO level. In order to enable DEBUG verbosity for DHCPv4 packet drops, you must create a configuration entry for the logger called "kea-dhcp4.bad-packets" and specify severity DEBUG for this logger. All other configuration parameters may be omitted for this logger if the logger should use the default values specified in the root logger's configuration.

If there are multiple logger specifications in the configuration that might match a particular logger, the specification with the more specific logger name takes precedence. For example, if there are entries for both "kea-dhcp4" and "kea-dhcp4.dhcpsrv", the main DHCPv4 server program — and all libraries it uses other than the dhcpsrv library (libdhcpsrv) — will log messages according to the configuration in the first entry ("kea-dhcp4"). Messages generated by the dhcpsrv library will be logged according to the configuration set by the second entry.

Currently defined loggers are defined in the following table. The "Software Package" column of this table specifies whether the particular loggers belong to the core Kea code (open source Kea binaries and libraries), or hook libraries (open source or premium).

Note that user-defined hook libraries should not use any of the loggers mentioned above, but should instead define new loggers with names that correspond to the libraries using them. Suppose that the user created the library called "libdhcp-packet-capture" to dump packets received and transmitted by the server to the file. An appropriate name for the logger could be **kea-dhcp4.packet-capture-hooks**. (Note that the hook library implementer only specifies the second part of this name, i.e. "packet-capture". The first part is a root-logger name and is prepended by the Kea logging system.) It is also important to note that since this new logger is a child of a root logger, it inherits the configuration from the root logger, something that can be overridden by an entry in the configuration file.

The easiest way to find a logger name is to configure all logging to go to a single destination and look for specific logger names. See Section 18.1.2 for details.

18.1.1.2 severity (string)

This specifies the category of messages logged. Each message is logged with an associated severity, which may be one of the following (in descending order of severity):

- FATAL - associated with messages generated by a condition that is so serious that the server cannot continue executing.
- ERROR - associated with messages generated by an error condition. The server will continue executing, but the results may not be as expected.
- WARN - indicates an out-of-the-ordinary condition. However, the server will continue executing normally.
- INFO - an informational message marking some event.
- DEBUG - messages produced for debugging purposes.



Logger Name	Software Package	Description
kea-ctrl-agent	core	The root logger for the Control Agent exposing the RESTful control API. All components used by the Control Agent inherit the settings from this logger.
kea-ctrl-agent.http	core	A logger which outputs log messages related to receiving, parsing, and sending HTTP messages.
kea-dhcp4	core	The root logger for the DHCPv4 server. All components used by the DHCPv4 server inherit the settings from this logger.
kea-dhcp6	core	The root logger for the DHCPv6 server. All components used by the DHCPv6 server inherit the settings from this logger.
kea-dhcp4.alloc-engine kea-dhcp6.alloc-engine	core	Used by the lease allocation engine, which is responsible for managing leases in the lease database, i.e. creating, modifying, and removing DHCP leases as a result of processing messages from the clients.
kea-dhcp4.bad-packets kea-dhcp6.bad-packets	core	Used by the DHCP servers for logging inbound client packets that were dropped or to which the server responded with a DHCPNAK. It allows administrators to configure a separate log output that contains only packet drop and reject entries.
kea-dhcp4.callouts kea-dhcp6.callouts	core	Used to log messages pertaining to the callouts registration and execution for the particular hook point.
kea-dhcp4.commands kea-dhcp6.commands	core	Used to log messages relating to the handling of commands received by the DHCP server over the command channel.
kea-dhcp4.database kea-dhcp6.database	core	Used to log messages relating to general operations on the relational databases and Cassandra.
kea-dhcp4.ddns kea-dhcp6.ddns	core	Used by the DHCP server to log messages related to the Client FQDN and Hostname option processing. It also includes log messages related to the relevant DNS updates.
kea-dhcp4.dhcp4	core	Used by the DHCPv4 server daemon to log basic operations.
kea-dhcp4.dhcpsrv kea-dhcp6.dhcpsrv	core	The base loggers for the libkea-dhcpsrv library.
kea-dhcp4.eval kea-dhcp6.eval	core	Used to log messages relating to the client classification expression evaluation code.
kea-dhcp4.host-cache-hooks kea-dhcp6.host-cache-hooks	libdhcp_host_cache premium hook library	This logger is used to log messages related to the operation of the Host Cache Hook Library.
kea-dhcp4.flex-id-hooks kea-dhcp6.flex-id-hooks	libdhcp_flex_id premium hook library	This logger is used to log messages related to the operation of the Flexible Identifiers Hook Library.
kea-dhcp4.ha-hooks kea-dhcp6.ha-hooks	libdhcp_ha hook library	This logger is used to log messages related to the operation of the High Availability Hook Library.
kea-dhcp4.hooks		Used to log messages related to the management of hooks libraries, e.g. registration and deregistration of the



When the severity of a logger is set to one of these values, it will only log messages of that severity and above (e.g. setting the logging severity to INFO will log INFO, WARN, ERROR, and FATAL messages). The severity may also be set to NONE, in which case all messages from that logger are inhibited.

Note

The `keactrl` tool, described in Chapter 6, can be configured to start the servers in verbose mode. If this is the case, the settings of the logging severity in the configuration file will have no effect, i.e. the servers will use a logging severity of DEBUG regardless of the logging settings specified in the configuration file. If you need to control severity via the configuration file, please make sure that the `kea_verbose` value is set to "no" within the `keactrl` configuration.

18.1.1.3 `debuglevel` (integer)

When a logger's severity is set to DEBUG, this value specifies what level of debug messages should be printed. It ranges from 0 (least verbose) to 99 (most verbose). If severity for the logger is not DEBUG, this value is ignored.

18.1.1.4 `output_options` (list)

Each logger can have zero or more `output_options`. These specify where log messages are sent and are explained in detail below.

18.1.1.4.1 `output` (string)

This value determines the type of output. There are several special values allowed here: **`stdout`** (messages are printed on standard output), **`stderr`** (messages are printed on stderr), **`syslog`** (messages are logged to syslog using default name), **`syslog:name`** (messages are logged to syslog using specified name). Any other value is interpreted as a filename to which messages should be written.

18.1.1.4.2 `flush` (true of false)

Flush buffers after each log message. Doing this will reduce performance but will ensure that if the program terminates abnormally, all messages up to the point of termination are output. The default is "true".

18.1.1.4.3 `maxsize` (integer)

Only relevant when the destination is a file. This is the maximum size in bytes that a log file may reach. When the maximum size is reached, the file is renamed and a new file opened. For example, a ".1" is appended to the name — if a ".1" file exists, it is renamed ".2", etc. This is referred to as rotation.

The default value is 10240000 (10MB). The smallest value that may be specified without disabling rotation is 204800. Any value less than this, including 0, disables rotation.

Note

Due to a limitation of the underlying logging library (`log4cplus`), rolling over the log files (from ".1" to ".2", etc) may show odd results; there can be multiple small files at the timing of roll over. This can happen when multiple processes try to roll over the files simultaneously. Version 1.1.0 of `log4cplus` solved this problem, so if this version or later of `log4cplus` is used to build Kea, the issue should not occur. Even for older versions, it is normally expected to happen rarely unless the log messages are produced very frequently by multiple different processes.

18.1.1.4.4 `maxver` (integer)

Only relevant when the destination is a file and rotation is enabled (i.e. `maxsize` is large enough). This is the maximum number of rotated versions that will be kept. Once that number of files has been reached, the oldest file, "log-name.maxver", will be discarded each time the log rotates. In other words, at most there will be the active log file plus `maxver` rotated files. The minimum and default value is 1.



18.1.1.5 Example Logger Configurations

In this example we want to set the Server logging to write to the console using standard output.

```
"Server": {
  "loggers": [
    {
      "name": "kea-dhcp4",
      "output_options": [
        {
          "output": "stdout"
        }
      ],
      "severity": "WARN"
    }
  ]
}
```

In this second example, we want to store debug log messages in a file that is at most 2MB and keep up to 8 copies of old logfiles. Once the logfile grows to 2MB, it will be renamed and a new file will be created.

```
"Server": {
  "loggers": [
    {
      "name": "kea-dhcp6",
      "output_options": [
        {
          "output": "/var/log/kea-debug.log",
          "maxver": 8,
          "maxsize": 204800,
          "flush": true
        }
      ],
      "severity": "DEBUG",
      "debuglevel": 99
    }
  ]
}
```

18.1.2 Logging Message Format

Each message written to the configured logging destinations comprises a number of components that identify the origin of the message and, if the message indicates a problem, information about the problem that may be useful in fixing it.

Consider the message below logged to a file:

```
2014-04-11 12:58:01.005 INFO [kea-dhcp4.dhcp4srv/27456]
  DHCP4SRV_MEMFILE_DB opening memory file lease database: type=memfile universe=4
```

Note: the layout of messages written to the system logging file (syslog) may be slightly different. This message has been split across two lines here for display reasons; in the logging file, it will appear on one line.

The log message comprises a number of components:

2014-04-11 12:58:01.005

The date and time at which the message was generated.

INFO

The severity of the message.

**[kea-dhcp4.dhcpsrv/27456]**

The source of the message. This includes two elements: the Kea process generating the message (in this case, **kea-dhcp4**) and the component within the program from which the message originated (**dhcpsrv**, which is the name of the common library used by DHCP server implementations). The number after the slash is a process id (pid).

DHCPSRV_MEMFILE_DB

The message identification. Every message in Kea has a unique identification, which can be used as an index into the *Kea Messages Manual* (https://jenkins.isc.org/job/Kea_doc/messages/kea-messages.html) from which more information can be obtained.

opening memory file lease database: type=memfile universe=4

A brief description. Within this text, information relating to the condition that caused the message to be logged will be included. In this example, the information is logged that the in-memory lease database backend will be used to store DHCP leases.

18.1.3 Logging During Kea Startup

The logging configuration is specified in the configuration file. However, when Kea starts, the file is not read until partway into the initialization process. Prior to that, the logging settings are set to default values, although it is possible to modify some aspects of the settings by means of environment variables. Note that in the absence of any logging configuration in the configuration file, the settings of the (possibly modified) default configuration will persist while the program is running.

The following environment variables can be used to control the behavior of logging during startup:

KEA_LOCKFILE_DIR

Specifies a directory where the logging system should create its lock file. If not specified, it is *prefix*/var/run/kea, where *prefix* defaults to /usr/local. This variable must not end with a slash. There is one special value: "none", which instructs Kea to not create a lock file at all. This may cause issues if several processes log to the same file.

KEA_LOGGER_DESTINATION

Specifies logging output. There are several special values.

stdout

Log to standard output.

stderr

Log to standard error.

syslog[:*fac*]

Log via syslog. The optional *fac* (which is separated from the word "syslog" by a colon) specifies the facility to be used for the log messages. Unless specified, messages will be logged using the facility "local0".

Any other value is treated as a name of the output file. If not specified otherwise, Kea will log to standard output.



Chapter 19

The Kea Shell

19.1 Overview

Kea 1.2.0 introduced the Control Agent (CA, see Chapter 7), which provides a RESTful control interface over HTTP. That API is typically expected to be used by various IPAMs and similar management systems. Nevertheless, there may be cases when an administrator wants to send a command to the CA directly. The Kea shell provides a way to do this. It is a simple command-line, scripting-friendly, text client that is able to connect to the CA, send it commands with parameters, retrieve the responses, and display them.

As the primary purpose of the Kea shell is as a tool in a scripting environment, it is not interactive. However, with simple tricks it can be run manually.

19.2 Shell Usage

kea-shell is run as follows:

```
kea-shell [--host hostname] [--port number] [--path path] [--timeout seconds] [--service ←  
service-name] [command]
```

where:

- **--host** *hostname* specifies the hostname of the CA. If not specified, "localhost" is used.
- **--port** *number* specifies the TCP port on which the CA listens. If not specified, 8000 is used.
- **--path** *path* specifies the path in the URL to connect to. If not specified, an empty path is used. As the CA listens at the empty path, this parameter is useful only with a reverse proxy.
- **--timeout** *seconds* specifies the timeout (in seconds) for the connection. If not given, 10 seconds is used.
- **--service** *service-name* specifies the target of a command. If not given, the CA will be used as the target. May be used more than once to specify multiple targets.
- **command** specifies the command to be sent. If not specified, the **list-commands** command is used.

Other switches are:

- **-h** prints a help message.
- **-v** prints the software version.



Once started, the shell reads parameters for the command from standard input, which are expected to be in JSON format. When all have been read, the shell establishes a connection with the CA using HTTP, sends the command, and awaits a response. Once that is received, it is displayed on standard output.

For a list of available commands, see Chapter 17; additional commands may be provided by hook libraries. If you are unsure which commands are supported, use the **list-commands** command. It will instruct the CA to return a list of all supported commands.

The following shows a simple example of usage:

```
$ kea-shell --host 192.0.2.1 --port 8001 --service dhcp4 list-commands
^D
```

After the command line is entered, the program waits for command parameters to be entered. Since **list-commands** does not take any arguments, CTRL-D (represented in the above example by "^D") is pressed to indicate end-of-file and terminate the parameter input. The shell then contacts the CA and prints out the list of available commands returned for the service named **dhcp4**.

It is envisaged that the Kea shell will be most frequently used in scripts; the next example shows a simple scripted execution. It sends the command "config-write" to the CA (the **--service** parameter hasn't been used), along with the parameters specified in param.json. The result will be stored in result.json.

```
$ cat param.json
"filename": "my-config-file.json"
$ cat param.json | kea-shell --host 192.0.2.1 config-write > result.json
```

When a reverse proxy is used to de-multiplex requests to different servers, the default empty path in the URL is not enough, so the **--path** parameter should be used. For instance, if requests to the "/kea" path are forwarded to the CA this can be used:

```
$ kea-shell --host 192.0.2.1 --port 8001 --path kea ...
```

Kea shell requires Python to be installed on the system. It has been tested with Python 2.7 and various versions of Python 3, up to 3.5. Since not every Kea deployment uses this feature and there are deployments that do not have Python, the Kea shell is not enabled by default. To use it, specify **--enable-shell** when running "configure" during the installation of Kea.

The Kea shell is intended to serve more as a demonstration of the RESTful interface's capabilities (and, perhaps, an illustration for people interested in integrating their management environments with Kea) than as a serious management client. Do not expect it to be significantly expanded in the future. It is, and will remain, a simple tool.



Chapter 20

YANG/NETCONF Support

20.1 Overview

Kea 1.5.0 introduced optional support for a YANG/NETCONF interface with the new **kea-netconf** NETCONF agent.

This bare-bones documentation is a work in progress. Its current purpose is to let engineers joining the project or perhaps advanced early adopters to get up to speed quickly.

20.2 Installing NETCONF

Note that to get its NETCONF capabilities, Kea uses Sysrepo, which has many dependencies. Unfortunately, some of them are not available as packages and need to be compiled manually.

Please note that building libyang requires a minimum gcc version of at least 4.9, so on some environments - like CentOS 7.5 - the system compiler cannot be used.

The following sections provide installation instructions for Ubuntu 18.04 and CentOS 7.5. Due to a more modern compiler and many available packages, the installation procedure is much simpler on Ubuntu.

20.2.1 Installing NETCONF on Ubuntu 18.04

For detailed installation instructions see the `Ubuntu installation notes` page.

20.2.2 Installing NETCONF on CentOS 7.5

For detailed installation instructions see the `CentOS installation notes` page.

CentOS 7.5's gcc compiler (version 4.8.5) is very old. Some sysrepo dependencies require at least version 4.9, which unfortunately means that a new compiler has to be installed. Also, many of the Sysrepo dependencies are not available in CentOS as packages, so for the time being they must be installed from sources.

20.3 Quick Sysrepo Overview

This section offers a rather brief overview of a subset of available functions in Sysrepo. For more complete information, see the Sysrepo homepage. You may also want to take a look at the `notes` made during a series of IETF Hackathons.



In YANG, configurations and state data are described in the YANG syntax in module files named: "**module-name**" ["**revision**"].yang

The revision part is optional and has YYYY-MM-DD format. An alternate XML syntax YIN is defined but less user-friendly. Top-level modules are named in Kea models (a short version of schema models).

List currently installed YANG modules:

```
$ sysrepectl -l
```

After installation the result should be similar to this:

```
Sysrepo schema directory: /home/thomson/devel/sysrepo-0.7.6/build/repository/yang/
Sysrepo data directory:  /home/thomson/devel/sysrepo-0.7.6/build/repository/data/
(Do not alter contents of these directories manually)
```

Module Name Submodules	Revision Enabled Features	Conformance	Data Owner	Permissions	↔
ietf-netconf-notifications	2012-02-06	Installed	root:root	666	↔
ietf-netconf	2011-06-01	Imported			↔
ietf-netconf-acm	2012-02-22	Imported			↔
nc-notifications	2008-07-14	Installed	root:root	666	↔
notifications	2008-07-14	Installed	root:root	666	↔
turing-machine	2013-12-27	Installed	root:root	666	↔
iana-if-type	2014-05-08	Installed			↔
ietf-interfaces	2014-05-08	Installed	root:root	666	↔
ietf-ip	2014-06-16	Installed			↔

There are two major modules that Kea is able to support: kea-dhcp4-server and kea-dhcp6-server. Note that while there is an active effort in the DHC working group at IETF to develop a DHCPv6 YANG model, a similar initiative in the past for DHCPv4 failed. Therefore, Kea uses its own dedicated models for DHCPv4 and DHCPv6 but partially supports the IETF model for DHCPv6. Those three models have extra modules as dependencies. The dependency modules are also provided in `src/share/yang/modules` in sources and `share/kea/yang/modules` after installation.

To install modules from sources, do the following:

```
cd src/share/yang/modules
sudo sysrepectl -i -s /home/thomson/devel/sysrepo-0.7.6/build/repository/yang -s . -g ietf- ↔
  dhcpv6-server*.yang
sudo sysrepectl -i -s /home/thomson/devel/sysrepo-0.7.6/build/repository/yang -s . -g kea- ↔
  dhcp4-server*.yang
sudo sysrepectl -i -s /home/thomson/devel/sysrepo-0.7.6/build/repository/yang -s . -g kea- ↔
  dhcp6-server*.yang
...
```

Note that the first `-s` parameter specifies the location of the YANG schema repository; you can check it with `sysrepectl -l`. This is a parameter that is configured during Sysrepo compilation and is detected by the Kea configuration under the `SYSREPO_REPO` name.

The installation should look similar to the following:



```

$ sudo sysrepoctl -i -s /home/thomson/devel/sysrepo-0.7.6/build/repository/yang -s . -g ←
  ietf-dhcpv6-server*.yang
Installing a new module from file 'ietf-dhcpv6-server@2018-11-20.yang'...
Installing the YANG file to '/home/thomson/devel/sysrepo-0.7.6/build/repository/yang/ietf- ←
  dhcpv6-server@2018-07-14.yang'...
Resolving dependency: 'ietf-dhcpv6-server' imports 'ietf-dhcpv6-options'...
Installing the YANG file to '/home/thomson/devel/sysrepo-0.7.6/build/repository/yang/ietf- ←
  dhcpv6-options@2018-07-14.yang'...
Resolving dependency: 'ietf-dhcpv6-options' imports 'ietf-dhcpv6-types'...
Installing the YANG file to '/home/thomson/devel/sysrepo-0.7.6/build/repository/yang/ietf- ←
  dhcpv6-types@2018-07-14.yang'...
Resolving dependency: 'ietf-dhcpv6-server' imports 'ietf-dhcpv6-types'...
Installing the YANG file to '/home/thomson/devel/sysrepo-0.7.6/build/repository/yang/ietf- ←
  dhcpv6-types@2018-07-14.yang'...
Resolving dependency: 'ietf-dhcpv6-server' imports 'ietf-interfaces'...
Schema of the module ietf-interfaces is already installed, skipping...
Installing data files for module 'ietf-dhcpv6-server'...
Resolving dependency: 'ietf-dhcpv6-server' imports 'ietf-dhcpv6-options'...
Skipping installation of data files for module 'ietf-dhcpv6-options'...
Resolving dependency: 'ietf-dhcpv6-options' imports 'ietf-dhcpv6-types'...
Skipping installation of data files for module 'ietf-dhcpv6-types'...
Resolving dependency: 'ietf-dhcpv6-server' imports 'ietf-dhcpv6-types'...
Skipping installation of data files for module 'ietf-dhcpv6-types'...
Resolving dependency: 'ietf-dhcpv6-server' imports 'ietf-interfaces'...
Installing data files for module 'ietf-interfaces'...
Notifying sysrepo about the change...
Install operation completed successfully.

```

You can confirm whether the models are imported correctly by using `sysrepoctl -l`

```

$ sysrepoctl -l
Sysrepo schema directory: /home/thomson/devel/sysrepo-0.7.6/build/repository/yang/
Sysrepo data directory: /home/thomson/devel/sysrepo-0.7.6/build/repository/data/
(Do not alter contents of these directories manually)

```

Module Name Submodules	Revision	Conformance	Data Owner	Permissions	←
	Enabled Features				
ietf-netconf-notifications	2012-02-06	Installed	root:root	666	←
ietf-netconf	2011-06-01	Imported			←
ietf-netconf-acm	2012-02-22	Imported			←
nc-notifications	2008-07-14	Installed	root:root	666	←
notifications	2008-07-14	Installed	root:root	666	←
turing-machine	2013-12-27	Installed	root:root	666	←
iana-if-type	2014-05-08	Installed			←
ietf-interfaces	2014-05-08	Installed	root:root	666	←
ietf-ip	2014-06-16	Installed			←
kea-dhcp4-server	2018-11-20	Installed	root:root	666	←
kea-dhcp6-server	2018-11-20	Installed	root:root	666	←



ietf-dhcpv6-server	2018-09-04	Installed	root:root	666	↔
ietf-dhcpv6-options	2018-09-04	Imported			↔
ietf-dhcpv6-types	2018-01-30	Imported			↔

To install a new revision of a module it must first be uninstalled, e.g. by:

```
sudo sysrepoctl -u -m kea-dhcp4-server
```

If the module is used (i.e. imported) by other modules, it can be uninstalled only after those modules have finished using it. Installation and uninstallation must be done in dependency order and reverse dependency order accordingly.

20.4 Supported YANG Models

The only currently supported models are **kea-dhcp4-server** and **kea-dhcp6-server**. There is partial support for **ietf-dhcpv6-server**, but the primary focus of testing was on Kea DHCP servers. Several other models (**kea-dhcp-ddns** and **kea-ctrl-agent**) are currently not supported.

20.5 Using the NETCONF Agent

The NETCONF agent follows this algorithm:

- For each managed server, get the initial configuration from the server through the control socket.
- Open a connection with the Sysrepo environment and establish two sessions with the startup and running datastores.
- Check that used (not essential) and required (essential) modules are installed in the sysrepo repository at the right revision. If an essential module, i.e. a module where the configuration schema for a managed server is defined, is not installed, raise a fatal error.
- For each managed server, get the YANG configuration from the startup datastore, translate it to JSON, and load it onto the server being configured.
- For each managed server, subscribe a module change callback using its model name.
- When a running configuration is changed, try to validate or load the updated configuration via the callback to the managed server.

20.6 Configuration

The behavior described in Section 20.5 is controlled by a few configuration flags, which can be set in the global scope or in a specific managed-server scope. In the second case, the value defined in the managed-server scope takes precedence. These flags are:

- The **boot-update** controls the initial configuration phase; when true (the default), the initial configuration retrieved from the classic Kea server JSON configuration file is loaded first, and then the startup YANG model is loaded. This setting lets administrators define a control socket in the local JSON file and then download the configuration from YANG. When set to false, this phase is skipped.
- The **subscribe-changes** command controls the module change subscription; when true (the default), a module change callback is subscribed, but when false the phase is skipped and running configuration updates are disabled. When set to true, the running datastore is used to subscribe for changes.



- The **validate-changes** command controls how Kea monitors changes in the Sysrepo configuration. Sysrepo offers two stages where Kea could interact: validation and application. At the validation (or SR_EV_VERIFY event, in the Sysrepo naming convention) stage, Kea retrieves the newly committed configuration and verifies it. If the configuration is incorrect for any reason, the Kea servers reject it and the error is propagated back to the Sysrepo, which then returns an error. This step only takes place if validate-changes is set to true. In the application (or SR_EV_APPLY event in the Sysrepo naming convention) stage, the actual configuration is applied. At this stage Kea can receive the configuration, but it is too late to signal back any errors as the configuration has already been committed.

The idea behind the initial configuration phase is to boot Kea servers with a minimal configuration which includes only a control socket, making them manageable. For instance, for the DHCPv4 server:

```
{
  "Dhcp4": {
    "control-socket": {
      "socket-type": "unix",
      "socket-name": "/tmp/kea4-sock"
    }
  }
}
```

Note the alternative to boot with full configurations does not allow easy tracking of changes or synchronization between the JSON and YANG configuration sources; therefore, that setup is not really compatible with the YANG / NETCONF configuration management paradigm, where everything should be performed in YANG.

With module change subscriptions enabled, the kea-netconf daemon will monitor any configuration changes as they appear in the Sysrepo. Such changes can be done using the **sysrepoctl** tool or remotely using any NETCONF client. For details, please see the Sysrepo documentation or Section 20.9. Those tools can be used to modify YANG configurations in the running datastore. Note that committed configurations are only updated in the running datastore; to keep them between server reboots they must be copied to the startup datastore.

When module changes are tracked (using **subscribe-changes** set to true) and the running configuration has changed (e.g. using **sysrepoctl** or any NETCONF client), the callback validates the modified configuration (if **validate-changes** was not set to false) and runs a second time to apply the new configuration. If the validation fails, the callback is still called again but with an ABORT (vs APPLY) event with rollback changes.

The returned code of the callback on an APPLY event is ignored, as it is too late to refuse a bad configuration.

There are four ways in which a modified YANG configuration could possibly be incorrect:

1. It can be non-compliant with the schema, e.g. unknown entry, missing mandatory entry, value with a bad type, or not matching a constraint.
2. It can fail to be translated from YANG to JSON, e.g. invalid user context.
3. It can fail Kea server sanity checks, e.g. out-of-subnet-pool range or unsupported database type.
4. The syntax is correct and passes server sanity checks but the configuration fails to run, e.g. the configuration specifies database credentials, but the database refuses connection.

The first case is handled by Sysrepo. The second and third cases are handled by kea-netconf in the validation phase (if not disabled by setting **validate-changes** to true). The last case causes the application phase to fail without a sensible report to Sysrepo.

The managed Kea servers or agents are described in the **managed-servers** section. Each sub-section begins by the service name: **dhcp4**, **dhcp6**, **d2** (the DHCP-DDNS server does not support the control channel feature yet), and **ca** (the control agent).

Each managed server entry contains optionally:

- **boot-update**, **subscribe-changes**, and **validate-changes** control flags.
- **model** specifies the YANG model / module name. For each service, the default is the corresponding Kea YANG model, e.g. for "**dhcp4**" it is "**kea-dhcp4-server**".



- **control-socket** specifies the control socket for managing the service configuration.

A control socket is specified by:

- **socket-type**: the socket type is either **stdout** (the default; it is not really a socket, but it allows **kea-netconf** to run in debugging mode where everything is printed on stdout. Can be also useful to redirect commands easily.), **unix** (standard direct server control channel, which uses UNIX sockets), and **http** (using a control agent, which accepts HTTP connections).
- **socket-name**: the local socket name for the **unix** socket type (default empty string).
- **socket-url**: the HTTP URL for the **http** socket type (default **http://127.0.0.1:8000/**).

User contexts can store arbitrary data as long as they are in valid JSON syntax and their top-level element is a map (i.e. the data must be enclosed in curly brackets). They are accepted at the NETCONF entry, i.e. below the top-level, managed-service entry, and control-socket entry scopes.

Hooks libraries can be loaded by the NETCONF agent just as with other servers or agents; however, currently no hook points are defined. The **hooks-libraries** list contains the list of hooks libraries that should be loaded by kea-netconf, along with their configuration information specified with **parameters**.

Please consult Chapter 18 for the details on how to configure logging. The NETCONF agent's root logger's name is **kea-netconf**, as given in the example above.

20.7 kea-netconf Configuration Example

The following example demonstrates the basic NETCONF configuration. More examples are available in the **doc/examples/netconf** directory in the Kea sources.

```
// This is a simple example of a configuration for the NETCONF agent.
// This server provides a YANG interface for all Kea servers and the agent.
{
  "Netconf":
  {
    // Control flags can be defined in the global scope or
    // in a managed server scope. Precedence are:
    // - use the default value (true)
    // - use the global value
    // - use the local value.
    // So this overwrites the default value:
    "boot-update": false,

    // This map specifies how each server is managed. For each server there
    // is a name of the YANG model to be used and the control channel.
    //
    // Currently three control channel types are supported:
    // "stdout" which outputs the configuration on the standard output,
    // "unix" which uses the local control channel supported by the
    // "dhcp4" and "dhcp6" servers ("d2" support is not yet available),
    // and "http" which uses the Control agent "ca" to manage itself or
    // to forward commands to "dhcp4" or "dhcp6".
    "managed-servers":
    {
      // This is how kea-netconf can communicate with the DHCPv4 server.
      "dhcp4":
      {
        "comment": "DHCP4 server",
        "model": "kea-dhcp4-server",
        "control-socket":
        {
          "socket-type": "unix",
```



```
        "socket-name": "/tmp/kea4-ctrl-socket"
    }
},

// DHCPv6 parameters.
"dhcp6":
{
    "model": "kea-dhcp6-server",
    "control-socket":
    {
        "socket-type": "unix",
        "socket-name": "/tmp/kea6-ctrl-socket"
    }
},

// Currently the DHCP-DDNS (nicknamed D2) server does not support
// a command channel.
"d2":
{
    "model": "kea-dhcp-ddns",
    "control-socket":
    {
        "socket-type": "stdout",
        "user-context": { "in-use": false }
    }
},

// Of course the Control Agent (nicknamed CA) supports HTTP.
"ca":
{
    "model": "kea-ctrl-agent",
    "control-socket":
    {
        "socket-type": "http",
        "socket-url": "http://127.0.0.1:8000/"
    }
}
},

// kea-netconf is able to load hooks libraries that augment its operation.
// Currently there are no hook points defined in kea-netconf
// processing.
"hooks-libraries": [
    // The hooks libraries list may contain more than one library.
    {
        // The only necessary parameter is the library filename.
        "library": "/opt/local/netconf-commands.so",

        // Some libraries may support parameters. Make sure you
        // type this section carefully, as kea-netconf does not
        // validate it (because the format is library-specific).
        "parameters": {
            "param1": "foo"
        }
    }
],

// Similar to other Kea components, Netconf also uses logging.
"loggers": [
    {
        "name": "kea-netconf",
        "output_options": [
```



```
        {
            "output": "/var/log/kea-netconf.log",
            // Several additional parameters are possible in
            // addition to the typical output.
            // Flush determines whether logger flushes output
            // to a file.
            // Maxsize determines maximum filesize before
            // the file is being rotated.
            // Maxver specifies the maximum number of
            // rotated files being kept.
            "flush": true,
            "maxsize": 204800,
            "maxver": 4
        },
        "severity": "INFO",
        "debuglevel": 0
    ]
}
}
```

20.8 Starting and Stopping the NETCONF Agent

kea-netconf accepts the following command-line switches:

- **-c *file*** - specifies the configuration file.
- **-d** - specifies whether the agent logging should be switched to debug/verbose mode. In verbose mode, the logging severity and debuglevel specified in the configuration file are ignored and "debug" severity and the maximum debuglevel (99) are assumed. The flag is convenient for temporarily switching the server into maximum verbosity, e.g. when debugging.
- **-t *file*** - specifies the configuration file to be tested. Kea-netconf attempts to load it and conducts sanity checks; note that certain checks are possible only while running the actual server. The actual status is reported with exit code (0 = configuration looks ok, 1 = error encountered). Kea will print out log messages to standard output and error to standard error when testing configuration.
- **-v** - displays the version of kea-netconf and exits.
- **-V** - displays the extended version information for kea-netconf and exits. The listing includes the versions of the libraries dynamically linked to Kea.
- **-W** - displays the Kea configuration report and exits. The report is a copy of the `config.report` file produced by `./configure`; it is embedded in the executable binary.

20.9 Step-by-Step NETCONF Agent Operation Example

Note

Copies of example configurations presented within this section can be found in the Kea source code, under `doc/examples/netconf/kea-dhcp6-operations`.



20.9.1 Setup of NETCONF Agent Operation Example

The test box has an Ethernet interface named eth1. On some systems it is possible to rename interfaces, for instance on a Linux with an ens38 interface:

```
# ip link set down dev ens38
# ip link set name eth1 dev ens38
# ip link set up dev eth1
```

The interface must have an address in the test prefix:

```
# ip -6 addr add 2001:db8::1/64 dev eth1
```

The Kea DHCPv6 server must be launched with the configuration specifying a control socket used to receive control commands. The **kea-netconf** process uses this socket to communicate with the DHCPv6 server, i.e. it pushes translated configurations to that server using control commands. The following is the example control socket specification for the Kea DHCPv6 server:

```
{
  "Dhcp6": {
    "control-socket": {
      "socket-type": "unix",
      "socket-name": "/tmp/kea6-sock"
    }
  }
}
```

In order to launch the Kea DHCPv6 server using the configuration contained within the `boot.json` file, run:

```
# kea-dhcp6 -d -c boot.json
```

The current configuration of the server can be fetched via control socket by running:

```
# echo '{ "command": "config-get" }' | socat UNIX:/tmp/kea6-sock '-,ignoreeof'
```

The following is the example `netconf.json` configuration for **kea-netconf**, to manage the Kea DHCPv6 server:

```
{
  "Netconf":
  {
    "managed-servers":
    {
      "dhcp6":
      {
        "control-socket":
        {
          "socket-type": "unix",
          "socket-name": "/tmp/kea6-sock"
        }
      }
    },
    "loggers":
    [
      {
        "name": "kea-netconf",
        "output_options":
        [
          {
            "output": "stderr"
          }
        ],
        "severity": "DEBUG",
      }
    ]
  }
}
```



```
        "debuglevel": 99
    }
  ]
}
}
```

Note that in production you should not need to log at the DEBUG level.

The Kea NETCONF agent is launched by:

```
# kea-netconf -d -c netconf.json
```

Now that both **kea-netconf** and **kea-dhcp6** are running, it is possible to populate updates to the configuration to the DHCPv6 server. The following is the configuration extracted from `startup.xml`:

```
<config xmlns="urn:ietf:params:xml:ns:yang:kea-dhcp6-server">
  <subnet6>
    <id>1</id>
    <pool>
      <start-address>2001:db8::1:0</start-address>
      <end-address>2001:db8::1:ffff</end-address>
      <prefix>2001:db8::1:0/112</prefix>
    </pool>
    <subnet>2001:db8::/64</subnet>
  </subnet6>
  <interfaces-config>
    <interfaces>eth1</interfaces>
  </interfaces-config>
  <control-socket>
    <socket-name>/tmp/kea6-sock</socket-name>
    <socket-type>unix</socket-type>
  </control-socket>
</config>
```

To populate this new configuration:

```
# sysrepoctl -d startup -f xml -i startup.xml kea-dhcp6-server
```

kea-netconf pushes the configuration found in the Sysrepo startup datastore to all Kea servers during its initialization phase, after it subscribes to module changes in the Sysrepo running datastore. This action copies the configuration from the startup datastore to the running datastore and enables the running datastore, making it available.

Changes to the running datastore are applied after validation to the Kea servers. Note that they are not by default copied back to the startup datastore, i.e. changes are not permanent.

20.9.2 Error Handling in NETCONF Operation Example

There are four classes of issues with the configurations applied via NETCONF:

1. The configuration does not comply with the YANG schema.
2. The configuration cannot be translated from YANG to the Kea JSON.
3. The configuration is rejected by the Kea server.
4. The configuration was validated by the Kea server but cannot be applied.

In the first case, consider the following `BAD-schema.xml` configuration file:



```
<config xmlns="urn:ietf:params:xml:ns:yang:kea-dhcp6-server">
  <subnet4>
    <id>1</id>
    <pool>
      <start-address>2001:db8::1:0</start-address>
      <end-address>2001:db8::1:ffff</end-address>
      <prefix>2001:db8::1:0/112</prefix>
    </pool>
    <subnet>2001:db8::/64</subnet>
  </subnet4>
  <interfaces-config>
    <interfaces>eth1</interfaces>
  </interfaces-config>
  <control-socket>
    <socket-name>/tmp/kea6-sock</socket-name>
    <socket-type>unix</socket-type>
  </control-socket>
</config>
```

is directly rejected by **sysrepcfg**:

```
# sysrepcfg -d running -f xml -i BAD-schema.xml kea-dhcp6-server
```

In the second case, the configuration is rejected by **kea-netconf**. For example, consider this **BAD-translator.xml** file:

```
<config xmlns="urn:ietf:params:xml:ns:yang:kea-dhcp6-server">
  <subnet6>
    <id>1</id>
    <pool>
      <start-address>2001:db8::1:0</start-address>
      <end-address>2001:db8::1:ffff</end-address>
      <prefix>2001:db8::1:0/112</prefix>
    </pool>
    <subnet>2001:db8::/64</subnet>
  </subnet6>
  <interfaces-config>
    <interfaces>eth1</interfaces>
  </interfaces-config>
  <control-socket>
    <socket-name>/tmp/kea6-sock</socket-name>
    <socket-type>unix</socket-type>
  </control-socket>
  <user-context>bad</user-context>
</config>
```

In the third case, the configuration is presented to the Kea DHCPv6 server and fails to validate as in this **BAD-config.xml** file:

```
<config xmlns="urn:ietf:params:xml:ns:yang:kea-dhcp6-server">
  <subnet6>
    <id>1</id>
    <pool>
      <start-address>2001:db8:1::0</start-address>
      <end-address>2001:db8:1::ffff</end-address>
      <prefix>2001:db8:1::0/112</prefix>
    </pool>
    <subnet>2001:db8::/64</subnet>
  </subnet6>
  <interfaces-config>
    <interfaces>eth1</interfaces>
  </interfaces-config>
```



```
<control-socket>
  <socket-name>/tmp/kea6-sock</socket-name>
  <socket-type>unix</socket-type>
</control-socket>
</config>
```

In the last case, the misconfiguration is detected too late and the change must be reverted in Sysrepo, e.g. using the startup datastore as a backup. For this reason, please use the **sysrepoconf** `--permanent / -p` option (or a similar feature of NETCONF clients) with care.

20.9.3 NETCONF Operation Example with Two Pools

This example adds a second pool to the initial (i.e. startup) configuration in the `twopools.xml` file:

```
<config xmlns="urn:ietf:params:xml:ns:yang:kea-dhcp6-server">
  <subnet6>
    <id>1</id>
    <pool>
      <start-address>2001:db8::1:0</start-address>
      <end-address>2001:db8::1:ffff</end-address>
      <prefix>2001:db8::1:0/112</prefix>
    </pool>
    <pool>
      <start-address>2001:db8::2:0</start-address>
      <end-address>2001:db8::2:ffff</end-address>
      <prefix>2001:db8::2:0/112</prefix>
    </pool>
  </subnet6>
  <interfaces-config>
    <interfaces>eth1</interfaces>
  </interfaces-config>
  <control-socket>
    <socket-name>/tmp/kea6-sock</socket-name>
    <socket-type>unix</socket-type>
  </control-socket>
</config>
```

This configuration is installed by:

```
# sysrepoconf -d running -f xml -i twopools.xml kea-dhcp6-server
```

20.9.4 NETCONF Operation Example with Two Subnets

This example specifies two subnets in the `twosubnets.xml` file:

```
<config xmlns="urn:ietf:params:xml:ns:yang:kea-dhcp6-server">
  <subnet6>
    <id>1</id>
    <pool>
      <start-address>2001:db8:1::</start-address>
      <end-address>2001:db8:1::ffff</end-address>
      <prefix>2001:db8:1::/112</prefix>
    </pool>
  </subnet6>
  <subnet6>
    <id>2</id>
    <pool>
```



```
<start-address>2001:db8:2::</start-address>
<end-address>2001:db8:2::ffff</end-address>
<prefix>2001:db8:2::/112</prefix>
</pool>
<subnet>2001:db8:2::/64</subnet>
</subnet6>
<interfaces-config>
  <interfaces>eth1</interfaces>
</interfaces-config>
<control-socket>
  <socket-name>/tmp/kea6-sock</socket-name>
  <socket-type>unix</socket-type>
</control-socket>
</config>
```

This configuration is installed by:

```
# sysrepcfg -d running -f xml -i twosubnets.xml kea-dhcp6-server
```

20.9.5 NETCONF Operation Example with Logging

This example adds a logger entry to the initial (i.e. startup) configuration in the logging.xml file:

```
<config xmlns="urn:ietf:params:xml:ns:yang:kea-dhcp6-server">
  <interfaces-config>
    <interfaces>eth1</interfaces>
  </interfaces-config>
  <subnet6>
    <id>1</id>
    <pool>
      <start-address>2001:db8::1:0</start-address>
      <end-address>2001:db8::1:ffff</end-address>
      <prefix>2001:db8::1:0/112</prefix>
    </pool>
    <subnet>2001:db8::/64</subnet>
  </subnet6>
  <control-socket>
    <socket-name>/tmp/kea6-sock</socket-name>
    <socket-type>unix</socket-type>
  </control-socket>
  <logger>
    <name>kea-dhcp6</name>
    <output-option>
      <output>stderr</output>
    </output-option>
    <debuglevel>99</debuglevel>
    <severity>DEBUG</severity>
  </logger>
</config>
```

The corresponding Kea configuration in JSON is:

```
{
  "Dhcp6": {
    "control-socket": {
      "socket-name": "/tmp/kea6-sock",
      "socket-type": "unix"
    },
    "interfaces-config": {
      "interfaces": [ "eth1" ]
    }
  }
}
```



```
},
"subnet6": [
  {
    "id": 1,
    "pools": [
      {
        "pool": "2001:db8::1:0/112"
      }
    ],
    "subnet": "2001:db8::/64"
  }
],
"loggers": [
  {
    "name": "kea-dhcp6",
    "output_options": [
      {
        "output": "stderr"
      }
    ],
    "severity": "DEBUG",
    "debuglevel": 99
  }
]
}
}
```

Finally, any of the previous examples can be replayed using **sysrepcfg** in edit mode as follows:

```
# sysrepcfg -d running -f xml -e vi kea-dhcp6-server
```

or, of course, using a NETCONF client like **netopeer2-cli** from the Netopeer2 NETCONF Toolset.



Appendix A

API Reference

Kea currently supports 130 commands: **build-report**, **cache-clear**, **cache-get**, **cache-get-by-id**, **cache-insert**, **cache-load**, **cache-remove**, **cache-size**, **cache-write**, **class-add**, **class-del**, **class-get**, **class-list**, **class-update**, **config-get**, **config-reload**, **config-set**, **config-test**, **config-write**, **dhcp-disable**, **dhcp-enable**, **ha-continue**, **ha-heartbeat**, **ha-scopes**, **ha-sync**, **lease4-add**, **lease4-del**, **lease4-get**, **lease4-get-all**, **lease4-update**, **lease4-wipe**, **lease6-add**, **lease6-bulk-apply**, **lease6-del**, **lease6-get**, **lease6-get-all**, **lease6-update**, **lease6-wipe**, **leases-reclaim**, **libreload**, **list-commands**, **network4-add**, **network4-del**, **network4-get**, **network4-list**, **network4-subnet-add**, **network4-subnet-del**, **network6-add**, **network6-del**, **network6-get**, **network6-list**, **network6-subnet-add**, **network6-subnet-del**, **remote-global-parameter4-del**, **remote-global-parameter4-get**, **remote-global-parameter4-get-all**, **remote-global-parameter4-set**, **remote-global-parameter6-del**, **remote-global-parameter6-get**, **remote-global-parameter6-get-all**, **remote-global-parameter6-set**, **remote-network4-del**, **remote-network4-get**, **remote-network4-list**, **remote-network4-set**, **remote-network6-del**, **remote-network6-get**, **remote-network6-list**, **remote-network6-set**, **remote-option-def4-del**, **remote-option-def4-get**, **remote-option-def4-get-all**, **remote-option-def4-set**, **remote-option-def6-del**, **remote-option-def6-get**, **remote-option-def6-get-all**, **remote-option-def6-set**, **remote-option4-global-del**, **remote-option4-global-get**, **remote-option4-global-get-all**, **remote-option4-global-set**, **remote-option6-global-del**, **remote-option6-global-get**, **remote-option6-global-get-all**, **remote-option6-global-set**, **remote-server4-del**, **remote-server4-get**, **remote-server4-get-all**, **remote-server4-set**, **remote-server6-del**, **remote-server6-get**, **remote-server6-get-all**, **remote-server6-set**, **remote-subnet4-del-by-id**, **remote-subnet4-del-by-prefix**, **remote-subnet4-get-by-id**, **remote-subnet4-get-by-prefix**, **remote-subnet4-list**, **remote-subnet4-set**, **remote-subnet6-del-by-id**, **remote-subnet6-del-by-prefix**, **remote-subnet6-get-by-id**, **remote-subnet6-get-by-prefix**, **remote-subnet6-list**, **remote-subnet6-set**, **reservation-add**, **reservation-del**, **reservation-get**, **reservation-get-all**, **reservation-get-page**, **shutdown**, **stat-lease4-get**, **stat-lease6-get**, **statistic-get**, **statistic-get-all**, **statistic-remove**, **statistic-remove-all**, **statistic-reset**, **statistic-reset-all**, **subnet4-add**, **subnet4-del**, **subnet4-get**, **subnet4-list**, **subnet4-update**, **subnet6-add**, **subnet6-del**, **subnet6-get**, **subnet6-list**, **subnet6-update**, **version-get**.

Commands supported by kea-ctrl-agent daemon: **build-report**, **config-get**, **config-reload**, **config-set**, **config-test**, **config-write**, **list-commands**, **shutdown**, **version-get**.

Commands supported by kea-dhcp-ddns daemon: **build-report**, **config-get**, **config-reload**, **config-set**, **config-test**, **config-write**, **list-commands**, **shutdown**, **version-get**.

Commands supported by kea-dhcp4 daemon: **build-report**, **cache-clear**, **cache-get**, **cache-get-by-id**, **cache-insert**, **cache-load**, **cache-remove**, **cache-size**, **cache-write**, **class-add**, **class-del**, **class-get**, **class-list**, **class-update**, **config-get**, **config-reload**, **config-set**, **config-test**, **config-write**, **dhcp-disable**, **dhcp-enable**, **ha-continue**, **ha-heartbeat**, **ha-scopes**, **ha-sync**, **lease4-add**, **lease4-del**, **lease4-get**, **lease4-get-all**, **lease4-update**, **lease4-wipe**, **leases-reclaim**, **libreload**, **list-commands**, **network4-add**, **network4-del**, **network4-get**, **network4-list**, **network4-subnet-add**, **network4-subnet-del**, **remote-global-parameter4-del**, **remote-global-parameter4-get**, **remote-global-parameter4-get-all**, **remote-global-parameter4-set**, **remote-network4-del**, **remote-network4-get**, **remote-network4-list**, **remote-network4-set**, **remote-option-def4-del**, **remote-option-def4-get**, **remote-option-def4-get-all**, **remote-option-def4-set**, **remote-option4-global-del**, **remote-option4-global-get**, **remote-option4-global-get-all**, **remote-option4-global-set**, **remote-server4-del**, **remote-server4-get**, **remote-server4-get-all**, **remote-server4-set**, **remote-subnet4-del-by-id**, **remote-subnet4-del-by-prefix**, **remote-subnet4-get-by-id**, **remote-subnet4-get-by-prefix**, **remote-subnet4-list**, **remote-subnet4-set**, **reservation-add**, **reservation-del**, **reservation-get**, **reservation-get-all**, **reservation-get-page**, **shutdown**, **stat-lease4-get**, **statistic-get**, **statistic-get-all**,



statistic-remove , **statistic-remove-all** , **statistic-reset** , **statistic-reset-all** , **subnet4-add** , **subnet4-del** , **subnet4-get** , **subnet4-list** , **subnet4-update** , **version-get** .

Commands supported by kea-dhcp6 daemon: **build-report** , **cache-clear** , **cache-get** , **cache-get-by-id** , **cache-insert** , **cache-load** , **cache-remove** , **cache-size** , **cache-write** , **class-add** , **class-del** , **class-get** , **class-list** , **class-update** , **config-get** , **config-reload** , **config-set** , **config-test** , **config-write** , **dhcp-disable** , **dhcp-enable** , **ha-continue** , **ha-heartbeat** , **ha-scopes** , **ha-sync** , **lease6-add** , **lease6-bulk-apply** , **lease6-del** , **lease6-get** , **lease6-get-all** , **lease6-update** , **lease6-wipe** , **leases-reclaim** , **libreload** , **list-commands** , **network6-add** , **network6-del** , **network6-get** , **network6-list** , **network6-subnet-add** , **network6-subnet-del** , **remote-global-parameter6-del** , **remote-global-parameter6-get** , **remote-global-parameter6-get-all** , **remote-global-parameter6-set** , **remote-network6-del** , **remote-network6-get** , **remote-network6-list** , **remote-network6-set** , **remote-option-def6-del** , **remote-option-def6-get** , **remote-option-def6-get-all** , **remote-option-def6-set** , **remote-option6-global-del** , **remote-option6-global-get** , **remote-option6-global-get-all** , **remote-option6-global-set** , **remote-server6-del** , **remote-server6-get** , **remote-server6-get-all** , **remote-server6-set** , **remote-subnet6-del-by-id** , **remote-subnet6-del-by-prefix** , **remote-subnet6-get-by-id** , **remote-subnet6-get-by-prefix** , **remote-subnet6-list** , **remote-subnet6-set** , **reservation-add** , **reservation-del** , **reservation-get** , **reservation-get-all** , **reservation-get-page** , **shutdown** , **stat-lease6-get** , **statistic-get** , **statistic-get-all** , **statistic-remove** , **statistic-remove-all** , **statistic-reset** , **statistic-reset-all** , **subnet6-add** , **subnet6-del** , **subnet6-get** , **subnet6-list** , **subnet6-update** , **version-get** .

Commands supported by cb_cmds hook library: **remote-global-parameter4-del** , **remote-global-parameter4-get** , **remote-global-parameter4-get-all** , **remote-global-parameter4-set** , **remote-global-parameter6-del** , **remote-global-parameter6-get** , **remote-global-parameter6-get-all** , **remote-global-parameter6-set** , **remote-network4-del** , **remote-network4-get** , **remote-network4-list** , **remote-network4-set** , **remote-network6-del** , **remote-network6-get** , **remote-network6-list** , **remote-network6-set** , **remote-option-def4-del** , **remote-option-def4-get** , **remote-option-def4-get-all** , **remote-option-def4-set** , **remote-option-def6-del** , **remote-option-def6-get** , **remote-option-def6-get-all** , **remote-option-def6-set** , **remote-option4-global-del** , **remote-option4-global-get** , **remote-option4-global-get-all** , **remote-option4-global-set** , **remote-option6-global-del** , **remote-option6-global-get** , **remote-option6-global-get-all** , **remote-option6-global-set** , **remote-server4-del** , **remote-server4-get** , **remote-server4-get-all** , **remote-server4-set** , **remote-server6-del** , **remote-server6-get** , **remote-server6-get-all** , **remote-server6-set** , **remote-subnet4-del-by-id** , **remote-subnet4-del-by-prefix** , **remote-subnet4-get-by-id** , **remote-subnet4-get-by-prefix** , **remote-subnet4-list** , **remote-subnet4-set** , **remote-subnet6-del-by-id** , **remote-subnet6-del-by-prefix** , **remote-subnet6-get-by-id** , **remote-subnet6-get-by-prefix** , **remote-subnet6-list** , **remote-subnet6-set** .

Commands supported by class_cmds hook library: **class-add** , **class-del** , **class-get** , **class-list** , **class-update** .

Commands supported by high_availability hook library: **ha-continue** , **ha-heartbeat** , **ha-scopes** , **ha-sync** .

Commands supported by host_cache hook library: **cache-clear** , **cache-get** , **cache-get-by-id** , **cache-insert** , **cache-load** , **cache-remove** , **cache-size** , **cache-write** .

Commands supported by host_cmds hook library: **reservation-add** , **reservation-del** , **reservation-get** , **reservation-get-all** , **reservation-get-page** .

Commands supported by lease_cmds hook library: **lease4-add** , **lease4-del** , **lease4-get** , **lease4-get-all** , **lease4-update** , **lease4-wipe** , **lease6-add** , **lease6-bulk-apply** , **lease6-del** , **lease6-get** , **lease6-get-all** , **lease6-update** , **lease6-wipe** .

Commands supported by stat_cmds hook library: **stat-lease4-get** , **stat-lease6-get** .

Commands supported by subnet_cmds hook library: **network4-add** , **network4-del** , **network4-get** , **network4-list** , **network4-subnet-add** , **network4-subnet-del** , **network6-add** , **network6-del** , **network6-get** , **network6-list** , **network6-subnet-add** , **network6-subnet-del** , **subnet4-add** , **subnet4-del** , **subnet4-get** , **subnet4-list** , **subnet4-update** , **subnet6-add** , **subnet6-del** , **subnet6-get** , **subnet6-list** , **subnet6-update** .

A.1 build-report reference

build-report - Returns a list of compilation options that this particular binary was built with

Supported by: **kea-dhcp4** , **kea-dhcp6** , **kea-dhcp-ddns** , **kea-ctrl-agent**

Availability: 1.2.0 (built-in)

Description and examples: See Section [17.3.1](#)

Command syntax:



```
{
  "command": "build-report"
}
```

Response syntax:

```
{
  "result": 0,
  "text": <string with build details>
}
```

A.2 cache-clear reference

cache-clear - This command removes all cached host reservations.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**host_cache** hook)

Description and examples: See Section [15.4.12.2](#)

Command syntax:

```
{
  "command": "cache-clear"
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.3 cache-get reference

cache-get - Returns full content of the host cache.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**host_cache** hook)

Description and examples: See Section [15.4.12.6](#)

Command syntax:

```
{
  "command": "cache-get"
}
```



Response syntax:

```
{
  "result": 0
  "text": "123 entries returned."
  "arguments": <list of host reservations>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.4 cache-get-by-id reference

cache-get-by-id - Returns entries matching the given identifier from the host cache.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.6.0 (**host_cache** hook)

Description and examples: See Section [15.4.12.7](#)

Command syntax:

```
{
  "command": "cache-get-by-id",
  "arguments": {
    "hw-address": "01:02:03:04:05:06"
  }
}
```

Response syntax:

```
{
  "result": 0
  "text": "2 entries returned."
  "arguments": <list of host reservations>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.5 cache-insert reference

cache-insert - This command may be used to manually insert a host into the cache.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**host_cache** hook)

Description and examples: See Section [15.4.12.8](#)

Command syntax:

```
{
  "command": "cache-insert",
  "arguments": {
    "hw-address": "01:02:03:04:05:06",
    "subnet-id4": 4,
    "subnet-id6": 0,
    "ip-address": "192.0.2.100",
    "hostname": "somehost.example.org",
    "client-classes4": [ ],
    "client-classes6": [ ],
    "option-data4": [ ],
    "option-data6": [ ],
    "next-server": "192.0.0.2",
    "server-hostname": "server-hostname.example.org",
    "boot-file-name": "bootfile.efi",
    "host-id": 0
  }
},
{
  "command": "cache-insert",
  "arguments": {
    "hw-address": "01:02:03:04:05:06",
    "subnet-id4": 0,
    "subnet-id6": 6,
    "ip-addresses": [ "2001:db8::cafe:babe" ],
    "prefixes": [ "2001:db8:dead:beef::/64" ],
    "hostname": "",
    "client-classes4": [ ],
    "client-classes6": [ ],
    "option-data4": [ ],
    "option-data6": [ ],
    "next-server": "0.0.0.0",
    "server-hostname": "",
    "boot-file-name": "",
    "host-id": 0
  }
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error



- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.6 cache-load reference

cache-load - This command allows load the contents of a file on disk into an in-memory cache.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**host_cache** hook)

Description and examples: See Section [15.4.12.5](#)

Command syntax:

```
{
  "command": "cache-load",
  "arguments": "/tmp/kea-host-cache.json"
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.7 cache-remove reference

cache-remove - The cache-remove command works similarly to reservation-get command.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**host_cache** hook)

Description and examples: See Section [15.4.12.9](#)

Command syntax:

```
{
  "command": "cache-remove",
  "arguments": {
    "ip-address": "192.0.2.1",
    "subnet-id": 123
  }
}
```

Another example that removes IPv6 host identifier by DUID and specific subnet-id is:

```
{
  "command": "cache-remove",
  "arguments": {
```



```
    "duid": "00:01:ab:cd:f0:a1:c2:d3:e4",  
    "subnet-id": 123  
  }  
}
```

Response syntax:

```
{  
  "result": <integer>,  
  "text": <string>  
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.8 cache-size reference

cache-size - Returns number of entries of the host cache.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.6.0 (**host_cache** hook)

Description and examples: See Section [15.4.12.3](#)

Command syntax:

```
{  
  "command": "cache-size"  
}
```

Response syntax:

```
{  
  "result": 0  
  "text": "123 entries."  
  "arguments": { "size": 123 }  
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.9 cache-write reference

cache-write - Instructs Kea to write its host cache content to disk.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**host_cache** hook)

Description and examples: See Section [15.4.12.4](#)

Command syntax:

```
{
  "command": "cache-write",
  "arguments": "/path/to/the/file.json"
}
```

The command takes one mandatory argument that specifies a filename path of a file to be written.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.10 class-add reference

class-add - This command is used to create and add a new class to the existing server configuration.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.5.0 (**class_cmds** hook)

Description and examples: See Section [15.4.7.1](#)

Command syntax:

```
{
  "command": "class-add",
  "arguments": {
    "client-classes": [ {
      "name": <name of the class>,
      "test": <test expression to be evaluated on incoming packets>,
      "option-data": [ <option values here> ],
      "option-def": [ <option definitions here> ],
      "next-server": <ipv4 address>,
      "server-hostname": <string>,
      "boot-file-name": <name of the boot file>
    } ]
  }
}
```



The **next-server**, **server-hostname** and **boot-file-name** are DHCPv4 specific. Only one client class can be added with a single command.

Response syntax:

```
{
  "result": 0,
  "text": "Class '<class-name>' added.",
}
```

The command will be successful (result 0), unless the class name is a duplicate or another error occurs (result 1).

A.11 class-del reference

class-del - This command is used to remove a client class from the server configuration.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.5.0 (**class_cmds** hook)

Description and examples: See Section [15.4.7.3](#)

Command syntax:

```
{
  "command": "class-del",
  "arguments": {
    "name": <name of the class>
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "Class '<class-name>' deleted."
}
```

The command will return a result of 3 (empty) if the client class doesn't exist. If the client class exists, the returned result is 0 if the deletion was successful and the result is 1 if the deletion is unsuccessful.

A.12 class-get reference

class-get - This command is used to return detailed information about an existing client class.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.5.0 (**class_cmds** hook)

Description and examples: See Section [15.4.7.5](#)

Command syntax:

```
{
  "command": "class-get",
  "arguments": {
    "name": <name of the class>
  }
}
```

Response syntax:



```
{
  "result": 0,
  "text": "Class '<class-name>' definition returned",
  "arguments": {
    "client-classes": [
      {
        "name": <name of the class>,
        "only-if-required": <only if required boolean value>,
        "test": <test expression to be evaluated on incoming packets>,
        "option-data": [ <option values here> ],
        "option-def": [ <option definitions here> ],
        "next-server": <ipv4 address>,
        "server-hostname": <string>,
        "boot-file-name": <name of the boot file>
      }
    ]
  }
}
```

The returned information depends on the DHCP server type, i.e. some parameters are specific to DHCPv4 server. Also, some parameters may not be returned if they are not set for the client class. If the class with specified name doesn't exist a result of 3 (empty) is returned. If the client class is found, the result of 0 is returned. If there is an error while processing the command, the result of 1 is returned.

A.13 class-list reference

class-list - This command is used to retrieve a list of all client classes from the server configuration.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.5.0 (**class_cmds** hook)

Description and examples: See Section [15.4.7.4](#)

Command syntax:

```
{
  "command": "class-list"
}
```

This command includes no arguments.

Response syntax:

```
{
  "result": 0,
  "text": "<number of> classes found",
  "arguments": {
    "client-classes": [
      {
        "name": <first class name>
      },
      {
        "name": <second class name>
      }
    ]
  }
}
```

The returned list of classes merely contains their names. In order to retrieve full information about one of these classes use Section [15.4.7.5](#). The returned result is 3 (empty) if no classes are found. If the command is processed successfully and the list



of client classes is not empty, the result of 0 is returned. If there is an error while processing the command, the result of 1 is returned.

A.14 class-update reference

class-update - This command is used to update an existing client class in the server configuration.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.5.0 (**class_cmds** hook)

Description and examples: See Section [15.4.7.2](#)

Command syntax:

```
{
  "command": "class-update",
  "arguments": {
    "client-classes": [ {
      "name": <name of the class>,
      "test": <test expression to be evaluated on incoming packets>,
      "option-data": [ <option values here> ],
      "option-def": [ <option definitions here> ],
      "next-server": <ipv4 address>,
      "server-hostname": <string>,
      "boot-file-name": <name of the boot file>
    } ]
  }
}
```

The **next-server**, **server-hostname** and **boot-file-name** are DHCPv4 specific. Only one client class can be updated with a single command.

Response syntax:

```
{
  "result": 0,
  "text": "Class '<class-name>' updated."
}
```

The command will return the result of 3 (empty) if the client class doesn't exist. If the client class exists, the returned result is 0 if the update was successful and the result is 1 if the update is unsuccessful.

A.15 config-get reference

config-get - Retrieves the current configuration used by the server. The configuration is roughly equal to the configuration file, but includes additional changes made by other commands and due to parameters inheritance.

Supported by: **kea-dhcp4**, **kea-dhcp6**, **kea-dhcp-ddns**, **kea-ctrl-agent**

Availability: 1.2.0 (built-in)

Description and examples: See Section [17.3.2](#)

Command syntax:

```
{
  "command": "config-get"
}
```



This command takes no parameters.

Response syntax:

```
{
  "result": <integer>,
  "arguments": {
    <JSON configuration here, starting with Dhcp4, Dhcp6, or Control-agent object>
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.16 config-reload reference

config-reload - The config-reload command instructs Kea to load again the configuration file that was used previously.

Supported by: **kea-dhcp4**, **kea-dhcp6**, **kea-dhcp-ddns**, **kea-ctrl-agent**

Availability: 1.2.0 (built-in)

Description and examples: See Section [17.3.3](#)

Command syntax:

```
{
  "command": "config-reload"
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.17 config-set reference

config-set - The config-set command instructs the server to replace its current configuration with the new configuration supplied in the command's arguments.

Supported by: **kea-dhcp4, kea-dhcp6, kea-dhcp-ddns, kea-ctrl-agent**

Availability: 1.2.0 (built-in)

Description and examples: See Section [17.3.9](#)

Command syntax:

```
{
  "command": "config-set",
  "arguments": {
    "<server>": {
    }
  }
}
```

where <server> is the configuration element name for a given server such as "Dhcp4" or "Dhcp6"

Response syntax:

```
{"result": 0, "text": "Configuration successful." }

or

{"result": 1, "text": "unsupported parameter: BOGUS (<string>:16:26)" }
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.18 config-test reference

config-test - The config-test command instructs the server to check whether the new configuration supplied in the command's arguments can be loaded.

Supported by: **kea-dhcp4, kea-dhcp6, kea-dhcp-ddns, kea-ctrl-agent**

Availability: 1.2.0 (built-in)

Description and examples: See Section [17.3.4](#)

Command syntax:

```
{
  "command": "config-test",
  "arguments": {
    "<server>": {
    }
  }
}
```



where `>server<` is the configuration element name for a given server such as "Dhcp4" or "Dhcp6"

Response syntax:

```
{"result": 0, "text": "Configuration seems sane..." }

or

{"result": 1, "text": "unsupported parameter: BOGUS (<string>:16:26)" }
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.19 config-write reference

config-write - The config-write command instructs the Kea server to write its current configuration to a file on disk.

Supported by: **kea-dhcp4**, **kea-dhcp6**, **kea-dhcp-ddns**, **kea-ctrl-agent**

Availability: 1.2.0 (built-in)

Description and examples: See Section [17.3.5](#)

Command syntax:

```
{
  "command": "config-write",
  "arguments": {
    "filename": "config-modified-2017-03-15.json"
  }
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.20 dhcp-disable reference

dhcp-disable - The dhcp-disable command globally disables the DHCP service.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (built-in)

Description and examples: See Section [17.3.11](#)

Command syntax:

```
{
  "command": "dhcp-disable",
  "arguments": {
    "max-period": 20
  }
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.21 dhcp-enable reference

dhcp-enable - The dhcp-enable command globally enables the DHCP service.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (built-in)

Description and examples: See Section [17.3.12](#)

Command syntax:

```
{
  "command": "dhcp-enable"
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.22 ha-continue reference

ha-continue - This command is used to resume the operation of the paused HA state machine.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**high_availability** hook)

Description and examples: See Section [15.4.9.13.3](#)

Command syntax:

```
{
  "command": "ha-continue"
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.23 ha-heartbeat reference

ha-heartbeat - This command is sent internally by Kea partner when operating in High Availability (HA) mode. It will retrieve the server HA state and clock value.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**high_availability** hook)

Description and examples: See Section [15.4.9.3 \[256\]](#)

Command syntax:

```
{
  "command": "ha-heartbeat",
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

The response to this command is different from the typical command response. The response will include server state (see Section [15.4.9.3](#) plus the current clock value.



A.24 ha-scopes reference

ha-scopes - This command modifies the scope that the server is responsible for serving when operating in High Availability (HA) mode.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**high_availability** hook)

Description and examples: See Section [15.4.9.13.2](#)

Command syntax:

```
{
  "command": "ha-scopes",
  "service": [ <service, typically "dhcp4" or "dhcp6"> ],
  "arguments": {
    "scopes": [ "HA_server1", "HA_server2" ]
  }
}
```

In the example given, the arguments configure the server to handle traffic from both HA_server1 and HA_server2 scopes.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.25 ha-sync reference

ha-sync - The command is issued to instruct the server running in HA mode to synchronize its local lease database with the selected peer.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.4.0 (**high_availability** hook)

Description and examples: See Section [15.4.9.13.1](#)

Command syntax:

```
{
  "command": "ha-sync",
  "service": [ <service affected: "dhcp4" or "dhcp6"> ],
  "arguments": {
    "server-name": <name of the partner server>,
    "max-period": <integer, in seconds>
  }
}
```

Response syntax:



```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.26 lease4-add reference

lease4-add - The lease4-add command adds a new IPv4 lease administratively.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (**lease_cmds** hook)

Description and examples: See Section [15.4.5.1](#)

Command syntax:

```
{
  "command": "lease4-add",
  "arguments": {
    "ip-address": "192.0.2.202",
    "hw-address": "1a:1b:1c:1d:1e:1f"
  }
}
```

Note that Kea 1.4 requires an additional argument, subnet-ID, which is optional as of Kea 1.5. A number of other more detailed optional arguments are also supported.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.27 lease4-del reference

lease4-del - lease4-del can be used to delete a lease from the lease database.

Supported by: **kea-dhcp4**

Availability: 1.3.0 ([lease_cmds](#) hook)

Description and examples: See Section [15.4.5.1.5](#)

Command syntax:

```
{
  "command": "lease4-del",
  "arguments": {
    "ip-address": "192.0.2.202"
  }
}
```

Specify the lease to be deleted either by IP address, or by identifier-type and identifier value. Currently supported identifiers are "hw-address" and "client-id".

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.28 lease4-get reference

lease4-get - lease4-get can be used to query the lease database and retrieve existing leases.

Supported by: **kea-dhcp4**

Availability: 1.3.0 ([lease_cmds](#) hook)

Description and examples: See Section [15.4.5.1.2](#)

Command syntax:

```
{
  "command": "lease4-get",
  "arguments": {
    "ip-address": "192.0.2.1"
  }
}
```

Response syntax:



```
{
  "arguments": {
    "client-id": "42:42:42:42:42:42:42:42",
    "cltt": 12345678,
    "fqdn-fwd": false,
    "fqdn-rev": true,
    "hostname": "myhost.example.com.",
    "hw-address": "08:08:08:08:08:08",
    "ip-address": "192.0.2.1",
    "state": 0,
    "subnet-id": 44,
    "valid-lft": 3600
  },
  "result": 0,
  "text": "IPv4 lease found."
}
```

lease4-get returns a result that indicates a result of the operation and lease details, if found. It has one of the following values: 0 (success), 1 (error) or 2 (empty).

A.29 lease4-get-all reference

lease4-get-all - lease4-get-all is used to retrieve all IPv4 leases or all leases for the specified set of subnets.

Supported by: **kea-dhcp4**

Availability: 1.4.0 ([lease_cmds](#) hook)

Description and examples: See Section [15.4.5.1.3](#)

Command syntax:

```
{
  "command": "lease4-get-all"
  "arguments": "subnets"
}
```

The lease4-get-all command may result in very large responses.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.30 lease4-update reference

lease4-update - The lease4-update command can be used to update existing leases.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (**lease_cmds** hook)

Description and examples: See Section [15.4.5.1.6](#)

Command syntax:

```
{
  "command": "lease4-update",
  "arguments": {
    "ip-address": "192.0.2.1",
    "hostname": "newhostname.example.org",
    "hw-address": "1a:1b:1c:1d:1e:1f",
    "subnet-id": 44,
    "force-create": true
  }
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.31 lease4-wipe reference

lease4-wipe - lease4-wipe is designed to remove all leases associated with a given subnet.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (**lease_cmds** hook)

Description and examples: See Section [15.4.5.1.7](#)

Command syntax:

```
{
  "command": "lease4-wipe",
  "arguments": {
    "subnet-id": 44
  }
}
```

Response syntax:



```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.32 lease6-add reference

lease6-add - The lease6-add command creates a new lease administratively.

Supported by: **kea-dhcp6**

Availability: 1.3.0 (**lease_cmds** hook)

Description and examples: See Section [15.4.5.1 \[212\]](#)

Command syntax:

```
{
  "command": "lease6-add",
  "arguments": {
    "subnet-id": 66,
    "ip-address": "2001:db8::3",
    "duid": "1a:1b:1c:1d:1e:1f:20:21:22:23:24",
    "iaid": 1234
  }
}
```

lease6-add can be also used to add leases for IPv6 prefixes..

Response syntax:

```
{ "result": 0, "text": "Lease added." }
{ "result": 1, "text": "missing parameter 'ip-address' (<string>:3:19)" }
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.33 lease6-bulk-apply reference

lease6-bulk-apply - The lease6-bulk-apply command can be used to create, update and delete multiple IPv6 leases in a single transaction. This is used to communicate lease changes between the HA peers but may be used in all cases when it is desired to apply multiple lease updates in a single transaction.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**lease_cmds** hook)

Description and examples: See Section [15.4.5.1.1](#)

Command syntax:

```
{
  "command": "lease6-bulk-apply",
  "arguments": {
    "deleted-leases": [
      {
        "ip-address": "2001:db8:abcd::",
        "type": "IA_PD",
        ...
      },
      {
        "ip-address": "2001:db8:abcd::234",
        "type": "IA_NA",
        ...
      }
    ],
    "leases": [
      {
        "subnet-id": 66,
        "ip-address": "2001:db8:cafe::",
        "type": "IA_PD",
        ...
      },
      {
        "subnet-id": 66,
        "ip-address": "2001:db8:abcd::333",
        "type": "IA_NA",
        ...
      }
    ]
  }
}
```

If any of the leases is malformed, all changes are rolled back. If the leases are well formed but the operation fails for one or more leases, the these leases are listed in the response but the changes are preserved for all leases for which the operation was successful. The "deleted-leases" and "leases" are optional parameters but one of them must be specified.

Response syntax:

```
{
  {
    "result": 0,
    "text": IPv6 leases bulk apply completed.
    "arguments": {
      "failed-deleted-leases": [
        {
          "ip-address": "2001:db8:abcd::",
          "type": "IA_PD",
          "result": <control result>,
          "error-message": <error message>
        }
      ]
    }
  }
}
```



```
    }
  ],
  "failed-leases": [
    {
      "ip-address": "2001:db8:cafe::",
      "type": "IA_PD",
      "result" <control result>,
      "error-message": <error message>
    }
  ]
}
}
```

The "failed-deleted-leases" holds the list of leases which failed to delete. This includes leases which were not found in the database. The "failed-leases" includes the list of leases which failed to create or update. For each lease for which there was an error while processing the lease, inserting it into the database etc. the result is set to 1. For each lease which was not deleted because the server didn't find it in the database the result of 3 is returned.

A.34 lease6-del reference

lease6-del - lease6-del can be used to delete a lease from the lease database.

Supported by: **kea-dhcp6**

Availability: 1.3.0 ([lease_cmds](#) hook)

Description and examples: See Section [15.4.5.1.5 \[220\]](#)

Command syntax:

```
{
  "command": "lease6-del",
  "arguments": {
    "ip-address": "192.0.2.202"
  }
}
```

lease6-del returns a result that indicates a outcome of the operation. It has one of the following values: 0 (success), 1 (error) or 3 (empty).

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.35 lease6-get reference

lease6-get - lease6-get can be used to query the lease database and retrieve existing leases.

Supported by: **kea-dhcp6**

Availability: 1.3.0 ([lease_cmds](#) hook)

Description and examples: See Section [15.4.5.1.2](#) [216]

Command syntax:

```
{
  "command": "lease6-get",
  "arguments": {
    "ip-address": "2001:db8:1234:ab::",
    "type": "IA_PD"
  }
}
```

lease6-get returns a result that indicates a result of the operation and lease details, if found. It has one of the following values: 0 (success), 1 (error) or 2 (empty).

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.36 lease6-get-all reference

lease6-get-all - lease6-get-all is used to retrieve all IPv6 leases or all leases for the specified set of subnets.

Supported by: **kea-dhcp6**

Availability: 1.3.0 ([lease_cmds](#) hook)

Description and examples: See Section [15.4.5.1.3](#) [217]

Command syntax:

```
{
  "command": "lease6-get-all",
  "arguments": {
    "subnets": [ 1, 2, 3, 4 ]
  }
}
```

Response syntax:



```
{
  "arguments": {
    "leases": [
      {
        "cltt": 12345678,
        "duid": "42:42:42:42:42:42:42:42",
        "fqdn-fwd": false,
        "fqdn-rev": true,
        "hostname": "myhost.example.com.",
        "hw-address": "08:08:08:08:08:08",
        "iaid": 1,
        "ip-address": "2001:db8:2::1",
        "preferred-lft": 500,
        "state": 0,
        "subnet-id": 44,
        "type": "IA_NA",
        "valid-lft": 3600
      },
      {
        "cltt": 12345678,
        "duid": "21:21:21:21:21:21:21:21",
        "fqdn-fwd": false,
        "fqdn-rev": true,
        "hostname": "",
        "iaid": 1,
        "ip-address": "2001:db8:0:0:2::",
        "preferred-lft": 500,
        "prefix-len": 80,
        "state": 0,
        "subnet-id": 44,
        "type": "IA_PD",
        "valid-lft": 3600
      }
    ]
  },
  "result": 0,
  "text": "2 IPv6 lease(s) found."
}
```

The lease6-get-all command may result in very large responses.

A.37 lease6-update reference

lease6-update - The lease6-update command can be used to update existing leases.

Supported by: **kea-dhcp6**

Availability: 1.3.0 ([lease_cmds](#) hook)

Description and examples: See Section [15.4.5.1.6](#) [220]

Command syntax:

```
{
  "command": "lease6-update",
  "arguments": {
    "ip-address": "2001:db8::1",
    "duid": "88:88:88:88:88:88:88:88",
    "iaid": 7654321,
    "hostname": "newhostname.example.org",
    "subnet-id": 66,
  }
}
```



```
    "force-create": false
  }
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.38 lease6-wipe reference

lease6-wipe - lease6-wipe is designed to remove all leases associated with a given subnet.

Supported by: **kea-dhcp6**

Availability: 1.3.0 (**lease_cmds** hook)

Description and examples: See Section [15.4.5.1.7 \[221\]](#)

Command syntax:

```
{
  "command": "lease6-wipe",
  "arguments": {
    "subnet-id": 66
  }
}
```

Note: not all backends support this command.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.39 leases-reclaim reference

leases-reclaim - The leases-reclaim command instructs the server to reclaim all expired leases immediately.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.0.0 (built-in)

Description and examples: See Section [17.3.6](#)

Command syntax:

```
{
  "command": "leases-reclaim",
  "arguments": {
    "remove": true
  }
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.40 libreload reference

libreload - The libreload command will first unload and then load all currently loaded hook libraries.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.2.0 (built-in)

Description and examples: See Section [17.3.7](#)

Command syntax:

```
{
  "command": "libreload",
  "arguments": { }
}
```

The server will respond with a result of 0 indicating success, or 1 indicating a failure.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:



- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.41 list-commands reference

list-commands - The list-commands command retrieves a list of all commands supported by the server.

Supported by: **kea-dhcp4**, **kea-dhcp6**, **kea-dhcp-ddns**, **kea-ctrl-agent**

Availability: 1.0.0 (built-in)

Description and examples: See Section [17.3.8](#)

Command syntax:

```
{
  "command": "list-commands",
  "arguments": { }
}
```

The server will respond with a list of all supported commands.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.42 network4-add reference

network4-add - The network4-add command is used to add a new shared network.

Supported by: **kea-dhcp4**

Availability: 1.3.0 ([subnet_cmds](#) hook)

Description and examples: See Section [15.4.6.13](#)

Command syntax:

```
{
  "command": "network4-add",
  "arguments": {
    "shared-networks": [ {
      "name": "floor13",
      "subnet4": [
```



```
{
  "id": 100,
  "pools": [ { "pool": "192.0.2.2-192.0.2.99" } ],
  "subnet": "192.0.2.0/24",
  "option-data": [
    {
      "name": "routers",
      "data": "192.0.2.1"
    }
  ]
},
{
  "id": 101,
  "pools": [ { "pool": "192.0.3.2-192.0.3.99" } ],
  "subnet": "192.0.3.0/24",
  "option-data": [
    {
      "name": "routers",
      "data": "192.0.3.1"
    }
  ]
} ]
} ]
}
```

Response syntax:

```
{
  "arguments": {
    "shared-networks": [ { "name": "floor13" } ]
  },
  "result": 0,
  "text": "A new IPv4 shared network 'floor13' added"
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.43 network4-del reference

network4-del - The network4-del command is used to delete existing shared networks.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (**subnet_cmds** hook)

Description and examples: See Section [15.4.6.14](#)

Command syntax:

```
{
  "command": "network4-del",
  "arguments": {
```



```
    "name": "floor13"
  }
}
```

Response syntax:

```
{
  "arguments": {
    "shared-networks": [
      {
        "name": "floor13"
      }
    ]
  },
  "result": 0,
  "text": "IPv4 shared network 'floor13' deleted"
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.44 network4-get reference

network4-get - The network4-get command is used to retrieve detailed information about shared networks, including subnets currently being part of a given network.

Supported by: **kea-dhcp4**

Availability: 1.3.0 ([subnet_cmds](#) hook)

Description and examples: See Section [15.4.6.12](#)

Command syntax:

```
{
  "command": "network4-get",
  "arguments": {
    "name": "floor13"
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "Info about IPv4 shared network 'floor13' returned",
  "arguments": {
    "shared-networks": [
      {
        "match-client-id": true,
        "name": "floor13",
        "option-data": [ ],
        "rebind-timer": 90,
        "relay": {
```



```
        "ip-address": "0.0.0.0"
      },
      "renew-timer": 60,
      "reservation-mode": "all",
      "subnet4": [
        {
          "subnet": "192.0.2.0/24",
          "id": 5,
          // many other subnet specific details here
        },
        {
          "id": 6,
          "subnet": "192.0.3.0/31",
          // many other subnet specific details here
        }
      ],
      "valid-lifetime": 120
    }
  ]
}
```

Note that the actual response contains many additional fields that are omitted here for clarity.

A.45 network4-list reference

network4-list - The network4-list command is used to retrieve full list of currently configured shared networks.

Supported by: **kea-dhcp4**

Availability: 1.3.0 ([subnet_cmds](#) hook)

Description and examples: See Section [15.4.6.11](#)

Command syntax:

```
{
  "command": "network4-list"
}
```

Response syntax:

```
{
  "arguments": {
    "shared-networks": [
      { "name": "floor1" },
      { "name": "office" }
    ]
  },
  "result": 0,
  "text": "2 IPv4 network(s) found"
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.46 network4-subnet-add reference

network4-subnet-add - The network4-subnet-add command is used to add existing subnets to existing shared networks.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (`subnet_cmds` hook)

Description and examples: See Section [15.4.6.15](#)

Command syntax:

```
{
  "command": "network4-subnet-add",
  "arguments": {
    "name": "floor13",
    "id": 5
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet 10.0.0.0/8 (id 5) is now part of shared network 'floor1'"
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.47 network4-subnet-del reference

network4-subnet-del - The network4-subnet-del command is used to remove a subnet that is part of an existing shared network and demote it to a plain, stand-alone subnet.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (`subnet_cmds` hook)

Description and examples: See Section [15.4.6.16](#)

Command syntax:

```
{
  "command": "network4-subnet-del",
  "arguments": {
    "name": "floor13",
    "id": 5
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet 10.0.0.0/8 (id 5) is now removed from shared network 'floor13'"
}
```



Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.48 network6-add reference

network6-add - The network6-add command is used to add a new shared network.

Supported by: **kea-dhcp6**

Availability: 1.3.0 ([subnet_cmds](#) hook)

Description and examples: See Section [15.4.6.13 \[230\]](#)

Command syntax:

```
{
  "command": "network4-add",
  "arguments": {
    "shared-networks": [ {
      "name": "floor13",
      "subnet4": [
        {
          "id": 100,
          "pools": [ { "pool": "192.0.2.2-192.0.2.99" } ],
          "subnet": "192.0.2.0/24",
          "option-data": [
            {
              "name": "routers",
              "data": "192.0.2.1"
            }
          ]
        }
      ]
    },
    {
      "id": 101,
      "pools": [ { "pool": "192.0.3.2-192.0.3.99" } ],
      "subnet": "192.0.3.0/24",
      "option-data": [
        {
          "name": "routers",
          "data": "192.0.3.1"
        }
      ]
    }
  ]
} ] ]
}
```

The network6-add uses the same syntax for both the query and the response. However, there are some parameters that are IPv4-only (e.g. match-client-id) and some are IPv6-only (e.g. interface-id).

Response syntax:

```
{
  "arguments": {
    "shared-networks": [ { "name": "floor13" } ]
  }
}
```



```
},
"result": 0,
"text": "A new IPv4 shared network 'floor13' added"
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.49 network6-del reference

network6-del - The network6-del command is used to delete existing shared networks.

Supported by: **kea-dhcp6**

Availability: 1.3.0 ([subnet_cmds](#) hook)

Description and examples: See Section [15.4.6.14 \[231\]](#)

Command syntax:

```
{
  "command": "network4-del",
  "arguments": {
    "name": "floor13"
  }
}
```

The network6-del command uses exactly the same syntax for both the command and the response.

Response syntax:

```
{
  "command": "network4-del",
  "arguments": {
    "name": "floor13",
    "subnets-action": "delete"
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.50 network6-get reference

network6-get - The network6-get command is used to retrieve detailed information about shared networks, including subnets currently being part of a given network.

Supported by: **kea-dhcp6**

Availability: 1.3.0 (`subnet_cmds` hook)

Description and examples: See Section [15.4.6.12 \[229\]](#)

Command syntax:

```
{
  "command": "network4-get",
  "arguments": {
    "name": "floor13"
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "Info about IPv4 shared network 'floor13' returned",
  "arguments": {
    "shared-networks": [
      {
        "match-client-id": true,
        "name": "floor13",
        "option-data": [ ],
        "rebind-timer": 90,
        "relay": {
          "ip-address": "0.0.0.0"
        },
        "renew-timer": 60,
        "reservation-mode": "all",
        "subnet4": [
          {
            "subnet": "192.0.2.0/24",
            "id": 5,
            // many other subnet specific details here
          },
          {
            "id": 6,
            "subnet": "192.0.3.0/31",
            // many other subnet specific details here
          }
        ],
        "valid-lifetime": 120
      }
    ]
  }
}
```

Note that the actual response contains many additional fields that are omitted here for clarity.

A.51 network6-list reference

network6-list - The network6-list command is used to retrieve full list of currently configured shared networks.



Supported by: **kea-dhcp6**

Availability: 1.3.0 (`subnet_cmds` hook)

Description and examples: See Section 15.4.6.11 [228]

Command syntax:

```
{
  "command": "network4-list"
}
```

`network6-list` follows exactly the same syntax for both the query and the response.

Response syntax:

```
{
  "arguments": {
    "shared-networks": [
      { "name": "floor1" },
      { "name": "office" }
    ]
  },
  "result": 0,
  "text": "2 IPv4 network(s) found"
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.52 network6-subnet-add reference

network6-subnet-add - The `network6-subnet-add` command is used to add existing subnets to existing shared networks.

Supported by: **kea-dhcp6**

Availability: 1.3.0 (`subnet_cmds` hook)

Description and examples: See Section 15.4.6.15 [231]

Command syntax:

```
{
  "command": "network4-subnet-add",
  "arguments": {
    "name": "floor13",
    "id": 5
  }
}
```

The `network6-subnet-add` command uses exactly the same syntax for both the command and the response.

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet 10.0.0.0/8 (id 5) is now part of shared network 'floor1'"
}
```



Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.53 network6-subnet-del reference

network6-subnet-del - The network6-subnet-del command is used to remove a subnet that is part of existing shared network and demote it to a plain, stand-alone subnet.

Supported by: **kea-dhcp6**

Availability: 1.3.0 (**subnet_cmds** hook)

Description and examples: See Section [15.4.6.16 \[232\]](#)

Command syntax:

```
{
  "command": "network4-subnet-del",
  "arguments": {
    "name": "floor13",
    "id": 5
  }
}
```

The network6-subnet-del command uses exactly the same syntax for both the command and the response.

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet 10.0.0.0/8 (id 5) is now removed from shared network 'floor13'"
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.54 remote-global-parameter4-del reference

remote-global-parameter4-del - This command is used to delete a global DHCPv4 parameter from the configuration database. The server will use the value specified in the configuration file or a default value (if the parameter is not specified in the configuration file) after deleting the parameter from the database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.6](#)

Command syntax:



```
{
  "command": "remote-global-parameter4-del",
  "arguments": {
    "parameters": [ <parameter name as string> ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

This command carries the list including exactly one name of the parameter to be deleted. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv4 global parameter(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.55 remote-global-parameter4-get reference

remote-global-parameter4-get - This command is used to fetch selected global parameter for the server from the specified database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.7](#)

Command syntax:

```
{
  "command": "remote-global-parameter4-get",
  "arguments": {
    "parameters": [ <parameter name as string> ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

This command carries a list including exactly one name of the parameter to be fetched. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed to fetch the global parameter value shared by all servers.

Response syntax:



```
{
  "result": 0,
  "text": "DHCPv4 global parameter found.",
  "arguments": {
    "parameters": {
      <parameter name>: <parameter value>,
      "metadata": {
        "server-tags": [ <server tag> ]
      }
    },
    "count": 1
  }
}
```

The returned response contains a map with a global parameter name/value pair. The value may be a JSON string, integer, real or boolean. The metadata is included and it provides database specific information associated with the returned object. If the "all" server tag was specified, the command attempts to fetch the global parameter value associated with all servers. If the explicit server tag is specified, the command will fetch the value associated with the given server. If the server specific value doesn't exist, it will try to fetch the value associated with all servers.

A.56 remote-global-parameter4-get-all reference

remote-global-parameter4-get-all - This command is used to fetch all global parameters for the server from the specified database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.8](#)

Command syntax:

```
{
  "command": "remote-global-parameter4-get-all",
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The special server tag "all" is allowed to fetch the global parameters shared by all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv4 global parameters found.",
  "arguments": {
    "parameters": [
      {
        <first parameter name>: <first parameter value>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      },
      {

```



```
        <second parameter name>: <second parameter value>,
        "metadata": {
            "server-tags": [ <server tag> ]
        }
    },
    "count": 2
}
}
```

The returned response contains a list of maps. Each map contains a global parameter name/value pair. The value may be a JSON string, integer, real or boolean. The metadata is appended to each parameter and it provides database specific information associated with the returned objects. If the server tag "all" is included in the command, the response contains the global parameters shared between all servers. It excludes server specific global parameters. If an explicit server tag is included in the command, the response contains all global parameters directly associated with the given server and the global parameters associated with all servers when server specific values are not present.

A.57 remote-global-parameter4-set reference

remote-global-parameter4-set - This command is used to create or update one more global parameters in the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.9](#)

Command syntax:

```
{
  "command": "remote-global-parameter4-set"
  "arguments": {
    "parameters": {
      <first parameter name>: <first parameter value>,
      <second parameter name>: <second parameter value>
    },
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

This command carries multiple global parameters with their values. Care should be taken when specifying more than one parameter because in some cases only a subset of the parameters may be successfully stored in the database and other parameters may fail to be stored. In such cases the **remote-global-parameter4-get-all** may be useful to verify the contents of the database after the update. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed and it associates the specified parameters with all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv4 global parameter(s) successfully set.",
  "arguments": {
    "parameters": {
      <first parameter name>: <first parameter value>,
      <second parameter name>: <second parameter value>
    },
  },
}
```



```
    "count": 2
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.58 remote-global-parameter6-del reference

remote-global-parameter6-del - This command is used to delete a global DHCPv6 parameter from the configuration database. The server will use the value specified in the configuration file or a default value (if the parameter is not specified in the configuration file) after deleting the parameter from the database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.6 \[240\]](#)

Command syntax:

```
{
  "command": "remote-global-parameter6-del",
  "arguments": {
    "parameters": [ <parameter name as string> ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

This command carries the list including exactly one name of the parameter to be deleted. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv6 global parameter(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.59 remote-global-parameter6-get reference

remote-global-parameter6-get - This command is used to fetch selected global parameter for the server from the specified database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.7 \[240\]](#)

Command syntax:

```
{
  "command": "remote-global-parameter6-get",
  "arguments": {
    "parameters": [ <parameter name as string> ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

This command carries a list including exactly one name of the parameter to be fetched. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed to fetch the global parameter value shared by all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv6 global parameter found.",
  "arguments": {
    "parameters": {
      <parameter name>: <parameter value>,
      "metadata": {
        "server-tags": [ <server tag> ]
      }
    },
    "count": 1
  }
}
```

The returned response contains a map with a global parameter name/value pair. The value may be a JSON string, integer, real or boolean. The metadata is included and it provides database specific information associated with the returned object. If the "all" server tag was specified, the command attempts to fetch the global parameter value associated with all servers. If the explicit server tag is specified, the command will fetch the value associated with the given server. If the server specific value doesn't exist, it will try to fetch the value associated with all servers.

A.60 remote-global-parameter6-get-all reference

remote-global-parameter6-get-all - This command is used to fetch all global parameters for the server from the specified database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.8 \[242\]](#)

Command syntax:



```
{
  "command": "remote-global-parameter6-get-all",
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The special server tag "all" is allowed to fetch the global parameters shared by all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv6 global parameters found.",
  "arguments": {
    "parameters": [
      {
        <first parameter name>: <first parameter value>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      },
      {
        <second parameter name>: <second parameter value>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      }
    ],
    "count": 2
  }
}
```

The returned response contains a list of maps. Each map contains a global parameter name/value pair. The value may be a JSON string, integer, real or boolean. The metadata is appended to each parameter and it provides database specific information associated with the returned objects. If the server tag "all" is included in the command, the response contains the global parameters shared between all servers. It excludes server specific global parameters. If an explicit server tag is included in the command, the response contains all global parameters directly associated with the given server and the global parameters associated with all servers when server specific values are not present.

A.61 remote-global-parameter6-set reference

remote-global-parameter6-set - This command is used to create or update one more global parameters in the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.9 \[243\]](#)

Command syntax:

```
{
  "command": "remote-global-parameter6-set"
  "arguments": {
```



```
"parameters": {
  <first parameter name>: <first parameter value>,
  <second parameter name>: <second parameter value>
},
"remote": {
  <specification of the database to connect to>
},
"server-tags": [ <single server tag as string> ]
}
```

This command carries multiple global parameters with their values. Care should be taken when specifying more than one parameter because in some cases only a subset of the parameters may be successfully stored in the database and other parameters may fail to be stored. In such cases the **remote-global-parameter6-get-all** may be useful to verify the contents of the database after the update. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed and it associates the specified parameters with all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv6 global parameter(s) successfully set.",
  "arguments": {
    "parameters": {
      <first parameter name>: <first parameter value>,
      <second parameter name>: <second parameter value>
    },
    "count": 2
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.62 remote-network4-del reference

remote-network4-del - This command is used to delete an IPv4 shared network from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.10](#)

Command syntax:

```
{
  "command": "remote-network4-del",
  "arguments": {
    "shared-networks": [
      {
        "name": <shared network name>
      }
    ]
  }
}
```



```
    ],
    "subnets-action": "keep" | "delete",
    "remote": {
        <specification of the database to connect to>
    }
}
```

This command includes a list with exactly one name of the shared network to be deleted. The **subnets-action** denotes whether the subnets in this shared network should be deleted or not. The **server-tags** parameter must not be specified for this command.

Response syntax:

```
{
  "result": 0,
  "text": "1 IPv4 shared network(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.63 remote-network4-get reference

remote-network4-get - This command is used to fetch selected IPv4 shared network for the server from the specified database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.11](#)

Command syntax:

```
{
  "command": "remote-network4-get"
  "arguments": {
    "shared-networks": [
      {
        "name": <shared network name>
      }
    ],
    "subnets-include": "full" | "no",
    "remote": {
        <specification of the database to connect to>
    }
  }
}
```

This command includes a list with exactly one name of the shared network to be returned. The **subnets-include** optional parameter allows for specifying whether the subnets belonging to the shared network should also be returned. The **server-tags** parameter must not be specified for this command.

Response syntax:



```
{
  "result": 0,
  "text": "IPv4 shared network found.",
  "arguments": {
    "shared-networks": [
      {
        "name": <shared network name>,
        "metadata": {
          "server-tags": [ <first server tag>, <second server tag>, ... ]
        },
        <the rest of the shared network information, potentially including subnets>
      }
    ],
    "count": 1
  }
}
```

If the subnets are returned with the shared network they are carried in the **subnet4** list within the shared network definition. The metadata is included in the returned shared network definition and it provides the database specific information associated with the returned object.

A.64 remote-network4-list reference

remote-network4-list - This command is used to fetch a list of all IPv4 shared networks from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.12](#)

Command syntax:

```
{
  "command": "remote-network4-list"
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <first server tag>, <second server tag>, ... ]
  }
}
```

The **server-tags** list is required for this command. This list must not be empty. It may either contain one or multiple server tags as strings or a single **null** value.

Response syntax:

```
{
  "result": 0,
  "text": "2 IPv4 shared network(s) found.",
  "arguments": {
    "shared-networks": [
      {
        "name": <first shared network name>,
        "metadata": {
          "server-tags": [ <first server tag>, <second server tag>, ... ]
        }
      },
      {
        "name": <second shared network name>,
```



```
        "metadata": {
            "server-tags": [ <first server tag>, ... ]
        }
    ],
    "count": 2
}
```

The returned response contains the list of maps. Each map contains the shared network name and the metadata which provides database specific information associated with the shared network. The returned list does not contain full definitions of the shared networks. Use **remote-network4-get** to fetch the full information about the selected shared networks. If the command includes explicit server tags as strings (including the special server tag "all"), the list contains all shared networks which are associated with any of the specified tags. A network is returned even if it is associated with multiple servers and only one of the specified tags matches. If the command includes the **null** value in the **server-tags** list, the response contains all shared networks which are assigned to no servers (unassigned).

A.65 remote-network4-set reference

remote-network4-set - This command is used to create or replace an IPv4 shared network in the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.13](#)

Command syntax:

```
{
  "command": "remote-network4-set",
  "arguments": {
    "shared-networks": [
      {
        <shared network specification excluding subnets list>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <first server tag>, <second server tag>, ... ]
  }
}
```

The provided list must contain exactly one shared network specification. It must not contain subnets ("subnet4" parameter). The subnets are added to the shared network using **remote-subnet4-set** command. The **server-tags** list is mandatory and it must contain one or more server tags as strings to explicitly associate the shared network with one or more user defined servers. It may include the special server tag "all" to associate the network with all servers.

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 shared network successfully set."
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error



- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.66 remote-network6-del reference

remote-network6-del - This command is used to delete an IPv6 shared network from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.10](#) [243]

Command syntax:

```
{
  "command": "remote-network6-del",
  "arguments": {
    "shared-networks": [
      {
        "name": <shared network name>
      }
    ],
    "subnets-action": "keep" | "delete",
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command includes a list with exactly one name of the shared network to be deleted. The **subnets-action** denotes whether the subnets in this shared network should be deleted or not. The **server-tags** parameter must not be specified for this command.

Response syntax:

```
{
  "result": 0,
  "text": "1 IPv6 shared network(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.67 remote-network6-get reference

remote-network6-get - This command is used to fetch selected IPv6 shared network for the server from the specified database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.11](#) [244]

Command syntax:



```
{
  "command": "remote-network6-get"
  "arguments": {
    "shared-networks": [
      {
        "name": <shared network name>
      }
    ],
    "subnets-include": "full" | "no",
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command includes a list with exactly one name of the shared network to be returned. The **subnets-include** optional parameter allows for specifying whether the subnets belonging to the shared network should also be returned. The **server-tags** parameter must not be specified for this command.

Response syntax:

```
{
  "result": 0,
  "text": "IPv6 shared network found.",
  "arguments": {
    "shared-networks": [
      {
        "name": <shared network name>,
        "metadata": {
          "server-tags": [ <first server tag>, <second server tag>, ... ]
        },
        <the rest of the shared network information, potentially including subnets>
      }
    ],
    "count": 1
  }
}
```

If the subnets are returned with the shared network they are carried in the **subnet6** list within the shared network definition. The metadata is included in the returned shared network definition and it provides the database specific information associated with the returned object.

A.68 remote-network6-list reference

remote-network6-list - This command is used to fetch a list of all IPv6 shared networks from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.12 \[244\]](#)

Command syntax:

```
{
  "command": "remote-network6-list"
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <first server tag>, <second server tag>, ... ]
  }
}
```



```
}  
}
```

The **server-tags** list is required for this command. This list must not be empty. It may either contain one or multiple server tags as strings or a single **null** value.

Response syntax:

```
{  
  "result": 0,  
  "text": "2 IPv6 shared network(s) found.",  
  "arguments": {  
    "shared-networks": [  
      {  
        "name": <first shared network name>,  
        "metadata": {  
          "server-tags": [ <first server tag>, <second server tag>, ... ]  
        }  
      },  
      {  
        "name": <second shared network name>,  
        "metadata": {  
          "server-tags": [ <first server tag>, ... ]  
        }  
      }  
    ],  
    "count": 2  
  }  
}
```

The returned response contains the list of maps. Each map contains the shared network name and the metadata which provides database specific information associated with the shared network. The returned list does not contain full definitions of the shared networks. Use **remote-network6-get** to fetch the full information about the selected shared networks. If the command includes explicit server tags as strings (including the special server tag "all"), the list contains all shared networks which are associated with any of the specified tags. A network is returned even if it is associated with multiple servers and only one of the specified tags matches. If the command includes the **null** value in the **server-tags** list, the response contains all shared networks which are assigned to no servers (unassigned).

A.69 remote-network6-set reference

remote-network6-set - This command is used to create or replace an IPv6 shared network in the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.13 \[245\]](#)

Command syntax:

```
{  
  "command": "remote-network6-set",  
  "arguments": {  
    "shared-networks": [  
      {  
        <shared network specification excluding subnets list>  
      }  
    ],  
    "remote": {  
      <specification of the database to connect to>  
    }  
  }  
}
```



```
    "server-tags": [ <first server tag>, <second server tag>, ... ]
  }
}
```

The provided list must contain exactly one shared network specification. It must not contain subnets ("subnet6" parameter). The subnets are added to the shared network using **remote-subnet6-set** command. The **server-tags** list is mandatory and it must contain one or more server tags as strings to explicitly associate the shared network with one or more user defined servers. It may include the special server tag "all" to associate the network with all servers.

Response syntax:

```
{
  "result": 0,
  "text": "IPv6 shared network successfully set."
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.70 remote-option-def4-del reference

remote-option-def4-del - This command is used to delete a DHCPv4 option definition from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.14](#)

Command syntax:

```
{
  "command": "remote-option-def4-del",
  "arguments": {
    "option-defs": [ {
      "code": <option code>,
      "space": <option space>
    } ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

This command includes a list with exactly one option definition specification comprising an option name and code. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error.

Response syntax:

```
{
  "result": 0,
  "text": "1 DHCPv4 option definition(s) deleted.",
  "arguments": {
```



```
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.71 remote-option-def4-get reference

remote-option-def4-get - This command is used to fetch a DHCPv4 option definition from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.15](#)

Command syntax:

```
{
  "command": "remote-option-def4-get",
  "arguments": {
    "option-defs": [
      {
        "code": <option code>,
        "space": <option space>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The desired option definition is identified by the pair of the option code/space values. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed to fetch the option definition instance shared by all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv4 option definition found.",
  "arguments": {
    "option-defs": [
      {
        <option definition>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      }
    ],
    "count": 1
  }
}
```



The metadata is included and it provides database specific information associated with the returned object. If the "all" server tag was specified, the command attempts to fetch the option definition associated with all servers. If the explicit server tag is specified, the command will fetch the option definition associated with the given server. If the server specific option definition doesn't exist, it will try to fetch the option definition associated with all servers.

A.72 remote-option-def4-get-all reference

remote-option-def4-get-all - This command is used to fetch all DHCPv4 option definitions from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.16](#)

Command syntax:

```
{
  "command": "remote-option-def4-get-all"
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The special server tag "all" is allowed to fetch the option definitions shared by all servers.

Response syntax:

```
{
  "result": 0,
  "text": "2 DHCPv4 option definition(s) found.",
  "arguments": {
    "option-defs": [
      {
        <first option definition>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      },
      {
        <second option definition>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      }
    ],
    "count": 2
  }
}
```

The returned response contains a list of maps. Each map contains an option definition specification and the metadata including database specific information associated with the returned objects. If the server tag "all" is included in the command, the response contains the option definitions shared between all servers. It excludes server specific option definitions. If an explicit server tag is included in the command, the response contains all option definitions directly associated with the given server and the option definitions associated with all servers when server specific option definitions are not present.



A.73 remote-option-def4-set reference

remote-option-def4-set - This command is used to create or replace DHCPv4 option definition in the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.17](#)

Command syntax:

```
{
  "command": "remote-option-def4-set",
  "arguments": {
    "option-defs": [
      {
        <option definition specification>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The provided list must contain exactly one option definition specification. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed and it associates the specified option definition with all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv4 option definition set."
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.74 remote-option-def6-del reference

remote-option-def6-del - This command is used to delete a DHCPv6 option definition from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.14](#) [246]

Command syntax:



```
{
  "command": "remote-option-def6-del",
  "arguments": {
    "option-defs": [ {
      "code": <option code>,
      "space": <option space>
    } ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

This command includes a list with exactly one option definition specification comprising an option name and code. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error.

Response syntax:

```
{
  "result": 0,
  "text": "1 DHCPv6 option definition(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.75 remote-option-def6-get reference

remote-option-def6-get - This command is used to fetch a DHCPv6 option definition from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.15 \[247\]](#)

Command syntax:

```
{
  "command": "remote-option-def6-get",
  "arguments": {
    "option-defs": [
      {
        "code": <option code>,
        "space": <option space>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```



```
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The desired option definition is identified by the pair of the option code/space values. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed to fetch the option definition instance shared by all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv6 option definition found.",
  "arguments": {
    "option-defs": [
      {
        <option definition>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      }
    ],
    "count": 1
  }
}
```

The metadata is included and it provides database specific information associated with the returned object. If the "all" server tag was specified, the command attempts to fetch the option definition associated with all servers. If the explicit server tag is specified, the command will fetch the option definition associated with the given server. If the server specific option definition doesn't exist, it will try to fetch the option definition associated with all servers.

A.76 remote-option-def6-get-all reference

remote-option-def6-get-all - This command is used to fetch all DHCPv6 option definitions from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.16](#) [247]

Command syntax:

```
{
  "command": "remote-option-def6-get-all"
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The special server tag "all" is allowed to fetch the option definitions shared by all servers.

Response syntax:

```
{
  "result": 0,
```



```
"text": "2 DHCPv6 option definition(s) found.",
"arguments": {
  "option-defs": [
    {
      <first option definition>,
      "metadata": {
        "server-tags": [ <server tag> ]
      }
    },
    {
      <second option definition>,
      "metadata": {
        "server-tags": [ <server tag> ]
      }
    }
  ],
  "count": 2
}
```

The returned response contains a list of maps. Each map contains an option definition specification and the metadata including database specific information associated with the returned objects. If the server tag "all" is included in the command, the response contains the option definitions shared between all servers. It excludes server specific option definitions. If an explicit server tag is included in the command, the response contains all option definitions directly associated with the given server and the option definitions associated with all servers when server specific option definitions are not present.

A.77 remote-option-def6-set reference

remote-option-def6-set - This command is used to create or replace DHCPv6 option definition in the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.17 \[248\]](#)

Command syntax:

```
{
  "command": "remote-option-def6-set",
  "arguments": {
    "option-defs": [
      {
        <option definition specification>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The provided list must contain exactly one option definition specification. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed and it associates the specified option definition with all servers.

Response syntax:

```
{
  "result": 0,
```



```
  "text": "DHCPv6 option definition set."  
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.78 remote-option4-global-del reference

remote-option4-global-del - This command is used to delete a DHCPv4 global option from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.18](#)

Command syntax:

```
{  
  "command": "remote-option4-global-del",  
  "arguments": {  
    "options": [  
      {  
        "code": <option code>  
        "space": <option space>  
      }  
    ],  
    "remote": {  
      <specification of the database to connect to>  
    },  
    "server-tags": [ <single server tag as string> ]  
  }  
}
```

This command includes a list with exactly one option specification comprising an option name and code. Specifying an empty list, a value of **null** or multiple server tags will result in an error.

Response syntax:

```
{  
  "result": 0,  
  "text": "1 DHCPv4 option(s) deleted.",  
  "arguments": {  
    "count": 1  
  }  
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.79 remote-option4-global-get reference

remote-option4-global-get - This command is used to fetch a global DHCPv4 option for the server from the specified database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.19](#)

Command syntax:

```
{
  "command": "remote-option4-global-get",
  "arguments": {
    "options": [
      {
        "code": <option code>,
        "space": <option space>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The option is identified by the pair of option code/space values. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed to fetch the global option instance shared by all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv4 option in found.",
  "arguments": {
    "options": [
      {
        <option information>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      }
    ]
  }
}
```

The metadata is included and it provides database specific information associated with the returned object. If the "all" server tag was specified, the command attempts to fetch the global option associated with all servers. If the explicit server tag is specified, the command will fetch the global option associated with the given server. If the server specific option doesn't exist, it will try to fetch the option associated with all servers.

A.80 remote-option4-global-get-all reference

remote-option4-global-get-all - This command is used to fetch all DHCPv4 global options for the server from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)



Description and examples: See Section [15.4.8.2.20](#)

Command syntax:

```
{
  "command": "remote-option4-global-get-all",
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The special server tag "all" is allowed to fetch the global options shared by all servers.

Response syntax:

```
{
  "result": 0,
  "text": "2 DHCPv4 option(s) found.",
  "arguments": {
    "options": [
      {
        <first option specification>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      },
      {
        <second option specification>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      }
    ],
    "count": 2
  }
}
```

The returned response contains a list of maps. Each map contains a global option specification and the metadata including database specific information associated with the returned object. If the server tag "all" is included in the command, the response contains the global options shared between all servers. It excludes server specific global options. If an explicit server tag is included in the command, the response contains all global options directly associated with the given server and the options associated with all servers when server specific options are not present.

A.81 remote-option4-global-set reference

remote-option4-global-set - This command is used to create or replace a DHCPv4 global option in the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.21](#)

Command syntax:

```
{
  "command": "remote-option4-global-set",
  "arguments": {
```



```
"options": [
  {
    <global option specification>
  }
],
"remote": {
  <specification of the database to connect to>
},
"server-tags": [ <single server tag as string> ]
}
```

The provided list must contain exactly one option specification. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed and it associates the specified option with all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv4 option set.",
  "arguments": {
    "options": [
      {
        "code": <option code>,
        "space": <option space>
      }
    ]
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.82 remote-option6-global-del reference

remote-option6-global-del - This command is used to delete a DHCPv6 global option from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.18 \[248\]](#)

Command syntax:

```
{
  "command": "remote-option6-global-del",
  "arguments": {
    "options": [
      {
        "code": <option code>
        "space": <option space>
      }
    ]
  }
}
```



```
"remote": {
    <specification of the database to connect to>
},
"server-tags": [ <single server tag as string> ]
}
```

This command includes a list with exactly one option specification comprising an option name and code. Specifying an empty list, a value of **null** or multiple server tags will result in an error.

Response syntax:

```
{
  "result": 0,
  "text": "1 DHCPv6 option(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.83 remote-option6-global-get reference

remote-option6-global-get - This command is used to fetch a global DHCPv6 option for the server from the specified database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.19 \[249\]](#)

Command syntax:

```
{
  "command": "remote-option6-global-get",
  "arguments": {
    "options": [
      {
        "code": <option code>,
        "space": <option space>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The option is identified by the pair of option code/space values. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed to fetch the global option instance shared by all servers.

Response syntax:



```
{
  "result": 0,
  "text": "DHCPv6 option in found.",
  "arguments": {
    "options": [
      {
        <option information>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      }
    ]
  }
}
```

The metadata is included and it provides database specific information associated with the returned object. If the "all" server tag was specified, the command attempts to fetch the global option associated with all servers. If the explicit server tag is specified, the command will fetch the global option associated with the given server. If the server specific option doesn't exist, it will try to fetch the option associated with all servers.

A.84 remote-option6-global-get-all reference

remote-option6-global-get-all - This command is used to fetch all DHCPv6 global options for the server from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.20 \[249\]](#)

Command syntax:

```
{
  "command": "remote-option6-global-get-all",
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The special server tag "all" is allowed to fetch the global options shared by all servers.

Response syntax:

```
{
  "result": 0,
  "text": "2 DHCPv6 option(s) found.",
  "arguments": {
    "options": [
      {
        <first option specification>,
        "metadata": {
          "server-tags": [ <server tag> ]
        }
      },
    ]
  }
}
```



```
        <second option specification>,
        "metadata": {
            "server-tags": [ <server tag> ]
        }
    ],
    "count": 2
}
}
```

The returned response contains a list of maps. Each map contains a global option specification and the metadata including database specific information associated with the returned object. If the server tag "all" is included in the command, the response contains the global options shared between all servers. It excludes server specific global options. If an explicit server tag is included in the command, the response contains all global options directly associated with the given server and the options associated with all servers when server specific options are not present.

A.85 remote-option6-global-set reference

remote-option6-global-set - This command is used to create or replace a DHCPv6 global option in the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.21 \[250\]](#)

Command syntax:

```
{
  "command": "remote-option6-global-set",
  "arguments": {
    "options": [
      {
        <global option specification>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <single server tag as string> ]
  }
}
```

The provided list must contain exactly one option specification. The **server-tags** list is mandatory and it must contain exactly one server tag. Specifying an empty list, a value of **null** or multiple server tags will result in an error. The server tag "all" is allowed and it associates the specified option with all servers.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv6 option set.",
  "arguments": {
    "options": [
      {
        "code": <option code>,
        "space": <option space>
      }
    ]
  }
}
```



Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.86 remote-server4-del reference

remote-server4-del - This command is used to delete information about a DHCPv4 server from the configuration database. Any configuration explicitly associated with the deleted server is automatically disassociated. In addition, configuration elements not shareable with other servers (e.g. global DHCP parameters) are deleted. The shareable configuration (e.g. subnets, shared networks) is not deleted as it may be used by other servers.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.2](#)

Command syntax:

```
{
  "command": "remote-server4-del",
  "arguments": {
    "servers": [
      {
        "server-tag": <server name>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command carries the list including exactly one map with the tag of the server to be deleted.

Response syntax:

```
{
  "result": 0,
  "text": "1 DHCPv4 server(s) deleted."
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.87 remote-server4-get reference

remote-server4-get - This command is used to fetch the information about the DHCPv4 server, such as server tag and description.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.3](#)

Command syntax:

```
{
  "command": "remote-server4-get",
  "arguments": {
    "servers": [
      {
        "server-tag": <server tag>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command carries the list including exactly one map with the tag of the server to be fetched.

Response syntax:

```
{
  "result": 0,
  "text": "DHCP server <server tag> found.",
  "arguments": {
    "servers": [
      {
        "server-tag": <server tag>,
        "description": <server description>
      }
    ],
    "count": 1
  }
}
```

The server tag is the unique identifier of the server, used to associate the configuration elements in the database with the particular server instance. The returned server description is specified by the user when setting the server information.

A.88 remote-server4-get-all reference

remote-server4-get-all - This command is used to fetch information about all DHCPv4 servers specified by the user.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.4](#)

Command syntax:

```
{
  "command": "remote-server4-get-all",
  "arguments": {
    "remote": {
```



```
    <specification of the database to connect to>
  }
}
```

This command contains no arguments besides the optional **remote**.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv4 servers found.",
  "arguments": {
    "servers": [
      {
        "server-tag": <first server tag>,
        "description": <first server description>
      },
      {
        "server-tag": <second server tag>,
        "description": <second server description>
      }
    ],
    "count": 2
  }
}
```

The returned response contain a list of maps. Each map contains a server tag uniquely identifying a server and the user defined description of the server. The Kea Configuration Backend uses the keyword **all** to associate parts of the configuration with all servers. Internally, it creates the logical server **all** for this purpose. However, this logical server is not returned as a result of the **remote-server4-get-all**. Only the user defined servers are returned.

A.89 remote-server4-set reference

remote-server4-set - This command is used to create or replace information about the DHCPv4 server in the database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.5](#)

Command syntax:

```
{
  "command": "remote-server4-set",
  "arguments": {
    "servers": [
      {
        "server-tag": <server tag>,
        "description": <server description>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

The provided list must contain exactly one server specification. The **server-tag** must be unique across all servers within the configuration database. The **description** is the arbitrary text describing the server, its location within the network etc.

Response syntax:



```
{
  "result": 0,
  "text": "DHCPv4 server successfully set.",
  "arguments": {
    "servers": [
      {
        "server-tag": <server tag>,
        "description": <server description>
      }
    ]
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.90 remote-server6-del reference

remote-server6-del - This command is used to delete information about a DHCPv6 server from the configuration database. Any configuration explicitly associated with the deleted server is automatically disassociated. In addition, configuration elements not shareable with other servers (e.g. global DHCP parameters) are deleted. The shareable configuration (e.g. subnets, shared networks) is not deleted as it may be used by other servers.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.2 \[237\]](#)

Command syntax:

```
{
  "command": "remote-server6-del",
  "arguments": {
    "servers": [
      {
        "server-tag": <server name>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command carries the list including exactly one map with the tag of the server to be deleted.

Response syntax:

```
{
  "result": 0,
  "text": "1 DHCPv6 server(s) deleted."
  "arguments": {
    "count": 1
  }
}
```



Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.91 remote-server6-get reference

remote-server6-get - This command is used to fetch the information about the DHCPv6 server, such as server tag and description.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.3 \[238\]](#)

Command syntax:

```
{
  "command": "remote-server6-get",
  "arguments": {
    "servers": [
      {
        "server-tag": <server tag>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command carries the list including exactly one map with the tag of the server to be fetched.

Response syntax:

```
{
  "result": 0,
  "text": "DHCP server <server tag> found.",
  "arguments": {
    "servers": [
      {
        "server-tag": <server tag>,
        "description": <server description>
      }
    ],
    "count": 1
  }
}
```

The server tag is the unique identifier of the server, used to associate the configuration elements in the database with the particular server instance. The returned server description is specified by the user when setting the server information.



A.92 remote-server6-get-all reference

remote-server6-get-all - This command is used to fetch information about all DHCPv6 servers specified by the user.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.4 \[239\]](#)

Command syntax:

```
{
  "command": "remote-server6-get-all",
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command contains no arguments besides the optional **remote**.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv6 servers found.",
  "arguments": {
    "servers": [
      {
        "server-tag": <first server tag>,
        "description": <first server description>
      },
      {
        "server-tag": <second server tag>,
        "description": <second server description>
      }
    ],
    "count": 2
  }
}
```

The returned response contain a list of maps. Each map contains a server tag uniquely identifying a server and the user defined description of the server. The Kea Configuration Backend uses the keyword **all** to associate parts of the configuration with all servers. Internally, it creates the logical server **all** for this purpose. However, this logical server is not returned as a result of the **remote-server6-get-all**. Only the user defined servers are returned.

A.93 remote-server6-set reference

remote-server6-set - This command is used to create or replace information about the DHCPv6 server in the database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.5 \[239\]](#)

Command syntax:



```
{
  "command": "remote-server6-set",
  "arguments": {
    "servers": [
      {
        "server-tag": <server tag>,
        "description": <server description>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

The provided list must contain exactly one server specification. The **server-tag** must be unique across all servers within the configuration database. The **description** is the arbitrary text describing the server, its location within the network etc.

Response syntax:

```
{
  "result": 0,
  "text": "DHCPv6 server successfully set.",
  "arguments": {
    "servers": [
      {
        "server-tag": <server tag>,
        "description": <server description>
      }
    ]
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.94 remote-subnet4-del-by-id reference

remote-subnet4-del-by-id - This command is used to delete an IPv4 subnet by ID from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.22](#)

Command syntax:

```
{
  "command": "remote-subnet4-del-by-id",
  "arguments": {
    "subnets": [
      {
        "id": <subnet identifier>
      }
    ]
  }
}
```



```
    }
  ],
  "remote": {
    <specification of the database to connect to>
  }
}
```

This command includes a list with exactly one id of the subnet to be deleted. The **server-tags** parameter must not be specified for this command.

Response syntax:

```
{
  "result": 0,
  "text": "1 IPv4 subnet(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.95 remote-subnet4-del-by-prefix reference

remote-subnet4-del-by-prefix - This command is used to delete an IPv4 subnet by prefix from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.23](#)

Command syntax:

```
{
  "command": "remote-subnet4-del-by-prefix",
  "arguments": {
    "subnets": [
      {
        "subnet": <subnet prefix>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command includes a list with exactly one prefix of the subnet to be deleted. The **server-tags** parameter must not be specified for this command.

Response syntax:



```
{
  "result": 0,
  "text": "1 IPv4 subnet(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.96 remote-subnet4-get-by-id reference

remote-subnet4-get-by-id - This command is used to fetch selected IPv4 subnet by ID for the server from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.24](#)

Command syntax:

```
{
  "command": "remote-subnet4-get-by-id"
  "arguments": {
    "subnets": [ {
      "id": <subnet identifier>
    } ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command includes a list with exactly one id of the subnet to be returned. The **server-tags** parameter must not be specified for this command.

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet found.",
  "arguments": {
    "subnets": [ {
      "id": <subnet identifier>,
      "subnet": <subnet prefix>,
      "shared-network-name": <shared network name> | null,
      "metadata": {
        "server-tags": [ <first server tag>, <second server tag>, ... ]
      },
      <the rest of the subnet specification here>
    } ],
  }
}
```



```
    "count": 1
  }
}
```

If the shared network name is null, it means that the returned subnet does not belong to any shared network (global subnet). The metadata is included in the returned subnet definition and it provides database specific information associated with the returned object.

A.97 remote-subnet4-get-by-prefix reference

remote-subnet4-get-by-prefix - This command is used to fetch selected IPv4 subnet by prefix from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.25](#)

Command syntax:

```
{
  "command": "remote-subnet4-get-by-prefix"
  "arguments": {
    "subnets": [ {
      "subnet": <subnet prefix>
    } ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command includes a list with exactly one prefix of the subnet to be returned. The **server-tags** parameter must not be specified for this command.

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet found.",
  "arguments": {
    "subnets": [
      {
        "id": <subnet identifier>,
        "subnet": <subnet prefix>,
        "shared-network-name": <shared network name> | null,
        "metadata": {
          "server-tags": [ <first server tag>, <second server tag>, ... ]
        },
        <the rest of the subnet specification here>
      }
    ],
    "count": 1
  }
}
```

If the shared network name is null, it means that the returned subnet does not belong to any shared network (global subnet). The metadata is included in the returned subnet definition and it provides database specific information associated with the returned object.



A.98 remote-subnet4-list reference

remote-subnet4-list - This command is used to fetch a list of all IPv4 subnets from the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.26](#)

Command syntax:

```
{
  "command": "remote-subnet4-list"
  "arguments": {
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <first server tag>, <second server tag>, ... ]
  }
}
```

The **server-tags** list is required for this command. This list must not be empty. It may either contain one or multiple server tags as strings or a single **null** value.

Response syntax:

```
{
  "result": 0,
  "text": "2 IPv4 subnets found.",
  "arguments": {
    "subnets": [
      {
        "id": <first subnet identifier>,
        "subnet": <first subnet prefix>,
        "shared-network-name": <shared network name> | null,
        "metadata": {
          "server-tags": [ <first server tag>, <second server tag>, ... ]
        }
      },
      {
        "id": <second subnet identifier>,
        "subnet": <second subnet prefix>,
        "shared-network-name": <shared network name> | null,
        "metadata": {
          "server-tags": [ <first server tag>, ... ]
        }
      }
    ],
    "count": 2
  }
}
```

The returned response contains a list of maps. Each map contains a subnet identifier, prefix and shared network name to which the subnet belongs. If the subnet does not belong to a shared network the name is null. The metadata includes database specific information associated with the subnets. The returned list does not contain full subnet definitions. Use **remote-subnet4-get** to fetch the full information about the selected subnets. If the command includes explicit server tags as strings (including the special server tag "all"), the list contains all subnets which are associated with any of the specified tags. A subnet is returned even if it is associated with multiple servers and only one of the specified tags matches. If the command includes the **null** value in the **server-tags** list, the response contains all subnets which are assigned to no servers (unassigned).



A.99 remote-subnet4-set reference

remote-subnet4-set - This command is used to create or replace an IPv4 subnet the configuration database.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.27](#)

Command syntax:

```
{
  "command": "remote-subnet4-set",
  "arguments": {
    "subnets": [
      {
        "id": <subnet identifier>,
        "subnet": <subnet prefix>,
        "shared-network-name": <shared network name> | null,
        <the rest of the subnet specification here>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <first server tag>, <second server tag>, ... ]
  }
}
```

The provided list must contain exactly one subnet specification. The **shared-network-name** parameter is required for these commands. It associates the subnet with the shared network by its name. If the subnet must not belong to any shared network (global subnet), the **null** value must be specified for the shared network name. The **server-tags** list is mandatory and it must contain one or more server tags as strings to explicitly associate the subnet with one or more user defined servers. It may include the special server tag "all" to associate the subnet with all servers.

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet successfully set.",
  "arguments": {
    "id": <subnet identifier>,
    "subnet": <subnet prefix>
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.100 remote-subnet6-del-by-id reference

remote-subnet6-del-by-id - This command is used to delete an IPv6 subnet by ID from the configuration database.



Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.22](#) [250]

Command syntax:

```
{
  "command": "remote-subnet6-del-by-id",
  "arguments": {
    "subnets": [
      {
        "id": <subnet identifier>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command includes a list with exactly one id of the subnet to be deleted. The **server-tags** parameter must not be specified for this command.

Response syntax:

```
{
  "result": 0,
  "text": "1 IPv6 subnet(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.101 remote-subnet6-del-by-prefix reference

remote-subnet6-del-by-prefix - This command is used to delete an IPv6 subnet by prefix from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.23](#) [251]

Command syntax:

```
{
  "command": "remote-subnet6-del-by-prefix",
  "arguments": {
    "subnets": [
      {
        "subnet": <subnet prefix>
      }
    ]
  }
}
```



```
    }
  ],
  "remote": {
    <specification of the database to connect to>
  }
}
```

This command includes a list with exactly one prefix of the subnet to be deleted. The **server-tags** parameter must not be specified for this command.

Response syntax:

```
{
  "result": 0,
  "text": "1 IPv6 subnet(s) deleted.",
  "arguments": {
    "count": 1
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.102 remote-subnet6-get-by-id reference

remote-subnet6-get-by-id - This command is used to fetch selected IPv6 subnet by ID for the server from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.24 \[251\]](#)

Command syntax:

```
{
  "command": "remote-subnet6-get-by-id"
  "arguments": {
    "subnets": [
      {
        "id": <subnet identifier>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command includes a list with exactly one id of the subnet to be returned. The **server-tags** parameter must not be specified for this command.

Response syntax:



```
{
  "result": 0,
  "text": "IPv6 subnet found.",
  "arguments": {
    "subnets": [
      {
        "id": <subnet identifier>,
        "subnet": <subnet prefix>,
        "shared-network-name": <shared network name> | null,
        "metadata": {
          "server-tags": [ <first server tag>, <second server tag>, ... ]
        },
        <the rest of the subnet specification here>
      }
    ],
    "count": 1
  }
}
```

If the shared network name is null, it means that the returned subnet does not belong to any shared network (global subnet). The metadata is included in the returned subnet definition and it provides database specific information associated with the returned object.

A.103 remote-subnet6-get-by-prefix reference

remote-subnet6-get-by-prefix - This command is used to fetch selected IPv6 subnet by prefix from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.25 \[251\]](#)

Command syntax:

```
{
  "command": "remote-subnet4-get-by-prefix"
  "arguments": {
    "subnets": [
      {
        "subnet": <subnet prefix>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    }
  }
}
```

This command includes a list with exactly one prefix of the subnet to be returned. The **server-tags** parameter must not be specified for this command.

Response syntax:

```
{
  "result": 0,
  "text": "IPv6 subnet found.",
  "arguments": {
    "subnets": [ {
      "id": <subnet identifier>,
      "subnet": <subnet prefix>,

```



```
    "shared-network-name": <shared network name> | null,  
    "metadata": {  
        "server-tags": [ <first server tag>, <second server tag>, ... ]  
    },  
    <the rest of the subnet specification here>  
} ],  
"count": 1  
}  
}
```

If the shared network name is null, it means that the returned subnet does not belong to any shared network (global subnet). The metadata is included in the returned subnet definition and it provides database specific information associated with the returned object.

A.104 remote-subnet6-list reference

remote-subnet6-list - This command is used to fetch a list of all IPv6 subnets from the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.26 \[252\]](#)

Command syntax:

```
{  
  "command": "remote-subnet6-list"  
  "arguments": {  
    "remote": {  
      <specification of the database to connect to>  
    },  
    "server-tags": [ <first server tag>, <second server tag>, ... ]  
  }  
}
```

The **server-tags** list is required for this command. This list must not be empty. It may either contain one or multiple server tags as strings or a single **null** value.

Response syntax:

```
{  
  "result": 0,  
  "text": "2 IPv6 subnets found.",  
  "arguments": {  
    "subnets": [  
      {  
        "id": <first subnet identifier>,  
        "subnet": <first subnet prefix>,  
        "shared-network-name": <shared network name> | null,  
        "metadata": {  
          "server-tags": [ <first server tag>, <second server tag>, ... ]  
        }  
      },  
      {  
        "id": <second subnet identifier>,  
        "subnet": <second subnet prefix>,  
        "shared-network-name": <shared network name> | null,  
        "metadata": {  
          "server-tags": [ <first server tag>, ... ]  
        }  
      }  
    ]  
  }  
}
```



```
    }
  ],
  "count": 2
}
}
```

The returned response contains a list of maps. Each map contains a subnet identifier, prefix and shared network name to which the subnet belongs. If the subnet does not belong to a shared network the name is null. The metadata includes database specific information associated with the subnets. The returned list does not contain full subnet definitions. Use **remote-subnet6-get** to fetch the full information about the selected subnets. If the command includes explicit server tags as strings (including the special server tag "all"), the list contains all subnets which are associated with any of the specified tags. A subnet is returned even if it is associated with multiple servers and only one of the specified tags matches. If the command includes the **null** value in the **server-tags** list, the response contains all subnets which are assigned to no servers (unassigned).

A.105 remote-subnet6-set reference

remote-subnet6-set - This command is used to create or replace an IPv6 subnet the configuration database.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (**cb_cmds** hook)

Description and examples: See Section [15.4.8.2.27 \[253\]](#)

Command syntax:

```
{
  "command": "remote-subnet6-set",
  "arguments": {
    "subnets": [
      {
        "id": <subnet identifier>,
        "subnet": <subnet prefix>,
        "shared-network-name": <shared network name> | null,
        <the rest of the subnet specification here>
      }
    ],
    "remote": {
      <specification of the database to connect to>
    },
    "server-tags": [ <first server tag>, <second server tag>, ... ]
  }
}
```

The provided list must contain exactly one subnet specification. The **shared-network-name** parameter is required for these commands. It associates the subnet with the shared network by its name. If the subnet must not belong to any shared network (global subnet), the **null** value must be specified for the shared network name. The **server-tags** list is mandatory and it must contain one or more server tags as strings to explicitly associate the subnet with one or more user defined servers. It may include the special server tag "all" to associate the subnet with all servers.

Response syntax:

```
{
  "result": 0,
  "text": "IPv6 subnet successfully set.",
  "arguments": {
    "id": <subnet identifier>,
    "subnet": <subnet prefix>
  }
}
```



Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.106 reservation-add reference

reservation-add - adds a new host reservation. The reservation may include IPv4 address, IPv6 addresses, IPv6 prefixes, various identifiers, a class the client will be assigned to, DHCPv4 and DHCPv6 options and more.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.2.0 (**host_cmds** hook)

Description and examples: See Section [15.4.4.2](#)

Command syntax:

```
{
  "command": "reservation-add",
  "arguments": {
    "reservation": {
      "boot-file-name": <string>,
      "comment": <string>,
      "client-id": <string>,
      "circuit-id": <string>,
      "duid": <string>,
      "flex-id": <string>,
      "ip-address": <string (IPv4 address)>,
      "ip-addresses": [ <comma separated strings> ],
      "hw-address": <string>,
      "hostname": <string>,
      "next-server": <string (IPv4 address)>,
      "option-data-list": [ <comma separated structures defining options> ],
      "prefixes": [ <comma separated IPv6 prefixes> ],
      "reservation-client-classes": [ <comma separated strings> ],
      "server-hostname": <string>,
      "subnet-id": <integer>,
      "user-context": <any valid JSON>,
    }
  }
}
```

Note the ip-address, client-id, next-server, server-hostname and boot-file-name are IPv4 specific. duid, ip-addresses and prefixes are IPv6 specific.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success



- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.107 reservation-del reference

reservation-del - Deletes an existing host reservation.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.2.0 (**host_cmds** hook)

Description and examples: See Section [15.4.4.6](#)

Command syntax:

```
{
  "command": "reservation-del",
  "arguments": {
    "subnet-id": <integer>,
    "ip-address": <string>,
    "identifier-type": <one of "hw-address", "duid", "circuit-id", "client-id" and " ←
      flex-id">,
    "identifier": <string>
  }
}
```

The host reservation can be identified by either (subnet-id, ip-address) pair or a triplet of (subnet-id, identifier-type, identifier).

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.108 reservation-get reference

reservation-get - Attempts to retrieve an existing host reservation

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.2.0 (**host_cmds** hook)

Description and examples: See Section [15.4.4.3](#)

Command syntax:



```
{
  "command": "reservation-get",
  "arguments": {
    "subnet-id": <integer>,
    "identifier-type": <string with one value out of: hw-address|duid|circuit-id|client ←
      -id|flex-id>,
    "identifier": <string>;
  }
}
```

The host reservation can be identified by either (subnet-id, ip-address) pair or a triplet of (subnet-id, identifier-type, identifier).

Response syntax:

```
{
  "result": <integer>,
  "text": <string>,
  "arguments": {
    "boot-file-name": <string>,
    "comment": <string>
    "client-id": <string>,
    "circuit-id": <string>,
    "duid": <string>,
    "flex-id": <string>,
    "ip-address": <string (IPv4 address)>,
    "ip-addresses": [ <comma separated strings> ],
    "hw-address": <string>,
    "hostname": <string>,
    "next-server": <string (IPv4 address)>,
    "option-data-list": [ <comma separated structures defining options> ],
    "prefixes": [ <comma separated IPv6 prefixes> ],
    "reservation-client-classes": [ <comma separated strings> ],
    "server-hostname": <string>,
    "subnet-id": <integer>,
    "user-context": <any valid JSON>,
  }
}
```

Arguments object appear only if a host is found. Many fields in the arguments object appear only if specific field is set.

A.109 reservation-get-all reference

reservation-get-all - Retrieve all host reservations for a specified subnet.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.6.0 (**host_cmds** hook)

Description and examples: See Section [15.4.4.4](#)

Command syntax:

```
{
  "command": "reservation-get-all",
  "arguments": {
    "subnet-id": <integer>
  }
}
```

Response syntax:



```
{
  "result": <integer>,
  "text": <string>
}
```

reservation-get-all command may result in very large responses.

A.110 reservation-get-page reference

reservation-get-page - Retrieve host reservations for a specified subnet by page.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.6.0 (**host_cmds** hook)

Description and examples: See Section [15.4.4.5](#)

Command syntax:

```
{
  "command": "reservation-get-page",
  "arguments": {
    "subnet-id": <integer>,
    "limit": <integer>,
    "source-index": <integer>,
    "from": <integer>
  }
}
```

the subnet id and the page size limit are mandatory. The source index and from host id are optional and default to 0. Values to use to next the next page are returned in responses in a next map.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.111 shutdown reference

shutdown - The shutdown command instructs the server to initiate its shutdown procedure.

Supported by: **kea-dhcp4**, **kea-dhcp6**, **kea-dhcp-ddns**, **kea-ctrl-agent**

Availability: 1.0.0 (built-in)

Description and examples: See Section [17.3.10](#)

Command syntax:



```
{
  "command": "shutdown"
}
```

The server will respond with a confirmation that the shutdown procedure has been initiated.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.112 stat-lease4-get reference

stat-lease4-get - The stat-lease4-get command fetches lease statistics for a range of known IPv4 subnets.

Supported by: **kea-dhcp4**

Availability: 1.4.0 ([stat_cmds](#) hook)

Description and examples: See Section [15.4.10.1](#)

Command syntax:

```
{
  "command": "stat-lease4-get"
}
```

Response syntax:

```
{
  "result": 0,
  "text": "stat-lease4-get: 2 rows found",
  "arguments": {
    "result-set": {
      "columns": [ "subnet-id",
                  "total-addresses",
                  "assigned-addresses",
                  "declined-addresses" ]
      "rows": [
        [ 10, 256, 111, 0 ],
        [ 20, 4098, 2034, 4 ]
      ],
      "timestamp": "2018-05-04 15:03:37.000000"
    }
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:



- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.113 stat-lease6-get reference

stat-lease6-get - The stat-lease6-get command fetches lease statistics for a range of known IPv6 subnets.

Supported by: **kea-dhcp6**

Availability: 1.4.0 (**stat_cmds** hook)

Description and examples: See Section [15.4.10.1 \[271\]](#)

Command syntax:

```
{
  "command": "stat-lease6-get",
  "arguments": {
    "subnet-id" : 10
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "stat-lease6-get: 2 rows found",
  "arguments": {
    "result-set": {
      "columns": [ "subnet-id", "total-nas", "assigned-nas", "declined-nas", "total-pds", ←
                  "assigned-pds" ]
      "rows": [
        [ 10, 4096, 2400, 3, 0, 0 ],
        [ 20, 0, 0, 0, 1048, 233 ]
        [ 30, 256, 60, 0, 1048, 15 ]
      ],
      "timestamp": "2018-05-04 15:03:37.000000"
    }
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.114 statistic-get reference

statistic-get - The statistic-get command retrieves a single statistic. It takes a single string parameter called name that specifies the statistic name.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.0.0 (built-in)

Description and examples: See Section [16.3.1](#)

Command syntax:

```
{
  "command": "statistic-get",
  "arguments": {
    "name": "pkt4-received"
  }
}
```

The server will respond with details of the requested statistic, with a result set to 0 indicating success and the specified statistic as the value of the "arguments" parameter.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.115 statistic-get-all reference

statistic-get-all - The statistic-get-all command retrieves all statistics recorded.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.0.0 (built-in)

Description and examples: See Section [16.3.4](#)

Command syntax:

```
{
  "command": "statistic-get-all",
  "arguments": { }
}
```

The server will respond with details of all recorded statistics, with result set to 0 indicating that it iterated over all statistics (even when the total number of statistics is zero).

Response syntax:



```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.116 statistic-remove reference

statistic-remove - The statistic-remove command attempts to delete a single statistic. It takes a single string parameter called name that specifies the statistic name.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.0.0 (built-in)

Description and examples: See Section [16.3.3](#)

Command syntax:

```
{
  "command": "statistic-remove",
  "arguments": {
    "name": "pkt4-received"
  }
}
```

If the specific statistic is found and its removal was successful, the server will respond with a status of 0, indicating success and an empty parameters field. If an error is encountered (e.g. requested statistic was not found), the server will return a status code of 1 (error) and the text field will contain the error description.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.117 statistic-remove-all reference

statistic-remove-all - The statistic-remove-all command attempts to delete all statistics.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.0.0 (built-in)

Description and examples: See Section [16.3.6](#)

Command syntax:

```
{
  "command": "statistic-remove-all",
  "arguments": { }
}
```

If the removal of all statistics was successful, the server will respond with a status of 0, indicating success and an empty parameters field. If an error is encountered, the server will return a status code of 1 (error) and the text field will contain the error description.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.118 statistic-reset reference

statistic-reset - The statistic-reset command sets the specified statistic to its neutral value: 0 for integer, 0.0 for float, 0h0m0s0us for time duration and "" for string type. It takes a single string parameter called name that specifies the statistic name.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.0.0 (built-in)

Description and examples: See Section [16.3.2](#)

Command syntax:

```
{
  "command": "statistic-reset",
  "arguments": {
    "name": "pkt4-received"
  }
}
```

If the specific statistic is found and reset was successful, the server will respond with a status of 0, indicating success and an empty parameters field. If an error is encountered (e.g. requested statistic was not found), the server will return a status code of 1 (error) and the text field will contain the error description.

Response syntax:



```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.119 statistic-reset-all reference

statistic-reset-all - The statistic-reset command sets all statistics to their neutral values: 0 for integer, 0.0 for float, 0h0m0s0us for time duration and "" for string type.

Supported by: **kea-dhcp4**, **kea-dhcp6**

Availability: 1.0.0 (built-in)

Description and examples: See Section [16.3.5](#)

Command syntax:

```
{
  "command": "statistic-reset-all",
  "arguments": { }
}
```

If the operation is successful, the server will respond with a status of 0, indicating success and an empty parameters field. If an error is encountered, the server will return a status code of 1 (error) and the text field will contain the error description.

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)



A.120 subnet4-add reference

subnet4-add - This command is used to create and add a new subnet to the existing server configuration.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (**subnet_cmds** hook)

Description and examples: See Section [15.4.6.5](#)

Command syntax:

```
{
  "command": "subnet4-add",
  "arguments": {
    "subnets": [ {
      "id": 123,
      "subnet": "10.20.30.0/24",
      ...
    } ]
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet added",
  "arguments": {
    "subnets": [
      {
        "id": 123,
        "subnet": "10.20.30.0/24"
      }
    ]
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.121 subnet4-del reference

subnet4-del - This command is used to remove a subnet from the server's configuration. This command has no effect on other configured subnets but removing a subnet has certain implications which the server's administrator should be aware of.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (**subnet_cmds** hook)

Description and examples: See Section [15.4.6.9](#)

Command syntax:



```
{
  "command": "subnet4-del",
  "arguments": {
    "id": 123
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet 192.0.2.0/24 (id 123) deleted",
  "arguments": {
    "subnets": [
      {
        "id": 123,
        "subnet": "192.0.2.0/24"
      }
    ]
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.122 subnet4-get reference

subnet4-get - This command is used to retrieve detailed information about the specified subnet. This command usually follows the `subnet4-list`, which is used to discover available subnets with their respective subnet identifiers and prefixes.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (`subnet_cmds` hook)

Description and examples: See Section [15.4.6.3](#)

Command syntax:

```
{
  "command": "subnet4-get",
  "arguments": {
    "id": 10
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "Info about IPv4 subnet 10.0.0.0/8 (id 10) returned",
  "arguments": {
    "subnets": [
      {
        "subnet": "10.0.0.0/8",

```



```
        "id": 1,
        "option-data": [
            ....
        ]
        ...
    }
]
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.123 subnet4-list reference

subnet4-list - This command is used to list all currently configured subnets. The subnets are returned in a brief form, i.e. a subnet identifier and subnet prefix is included for each subnet.

Supported by: **kea-dhcp4**

Availability: 1.3.0 (**subnet_cmds** hook)

Description and examples: See Section [15.4.6.1](#)

Command syntax:

```
{
  "command": "subnet4-list"
}
```

Response syntax:

```
{
  "result": 0,
  "text": "2 IPv4 subnets found",
  "arguments": {
    "subnets": [
      {
        "id": 10,
        "subnet": "10.0.0.0/8"
      },
      {
        "id": 100,
        "subnet": "192.0.2.0/24"
      }
    ]
  }
}
```

If no IPv4 subnets are found, an error code is returned along with the error description.



A.124 subnet4-update reference

subnet4-update - This command is used to update a subnet in the existing server configuration.

Supported by: **kea-dhcp4**

Availability: 1.6.0 (**subnet_cmds** hook)

Description and examples: See Section [15.4.6.7](#)

Command syntax:

```
{
  "command": "subnet4-update",
  "arguments": {
    "subnets": [ {
      "id": 123,
      "subnet": "10.20.30.0/24",
      ...
    } ]
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "IPv4 subnet updated",
  "arguments": {
    "subnets": [
      {
        "id": 123,
        "subnet": "10.20.30.0/24"
      }
    ]
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.125 subnet6-add reference

subnet6-add - This command is used to create and add new subnet to the existing server configuration. This operation has no impact on other subnets.

Supported by: **kea-dhcp6**

Availability: 1.3.0 (**subnet_cmds** hook)

Description and examples: See Section [15.4.6.6](#)

Command syntax:



```
{
  "command": "subnet6-add",
  "arguments": {
    "subnet6": [ {
      "id": 234,
      "subnet": "2001:db8:1::/64",
      ...
    } ]
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "IPv6 subnet added",
  "arguments": {
    "subnet6": [
      {
        "id": 234,
        "subnet": "2001:db8:1::/64"
      }
    ]
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.126 subnet6-del reference

subnet6-del - This command is used to remove a subnet from the server's configuration. This command has no effect on other configured subnets but removing a subnet has certain implications which the server's administrator should be aware of.

Supported by: **kea-dhcp6**

Availability: 1.3.0 (**subnet_cmds** hook)

Description and examples: See Section [15.4.6.10](#)

Command syntax:

```
{
  "command": "subnet6-del",
  "arguments": {
    "id": 234
  }
}
```

Response syntax:



```
{
  "result": 0,
  "text": "IPv6 subnet 2001:db8:1::/64 (id 234) deleted",
  "subnets": [
    {
      "id": 234,
      "subnet": "2001:db8:1::/64"
    }
  ]
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.127 subnet6-get reference

subnet6-get - This command is used to retrieve detailed information about the specified subnet. This command usually follows the `subnet6-list`, which is used to discover available subnets with their respective subnet identifiers and prefixes.

Supported by: **kea-dhcp6**

Availability: 1.3.0 (`subnet_cmds` hook)

Description and examples: See Section [15.4.6.4](#)

Command syntax:

```
{
  "command": "subnet6-get",
  "arguments": {
    "id": 11
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "Info about IPv6 subnet 2001:db8:1::/64 (id 11) returned",
  "arguments": {
    "subnets": [
      {
        "subnet": "2001:db8:1::/64",
        "id": 1,
        "option-data": [
          ...
        ]
        ....
      }
    ]
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:



- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.128 subnet6-list reference

subnet6-list - This command is used to list all currently configured subnets. The subnets are returned in a brief form, i.e. a subnet identifier and subnet prefix is included for each subnet.

Supported by: **kea-dhcp6**

Availability: 1.3.0 (`subnet_cmds` hook)

Description and examples: See Section [15.4.6.2](#)

Command syntax:

```
{
  "command": "subnet6-list"
}
```

Response syntax:

```
{
  "result": 0,
  "text": "2 IPv6 subnets found",
  "arguments": {
    "subnets": [
      {
        "id": 11,
        "subnet": "2001:db8:1::/64"
      },
      {
        "id": 233,
        "subnet": "3000::/16"
      }
    ]
  }
}
```

If no IPv6 subnets are found, an error code is returned along with the error description.

A.129 subnet6-update reference

subnet6-update - This command is used to update a subnet in the existing server configuration. This operation has no impact on other subnets.

Supported by: **kea-dhcp6**

Availability: 1.6.0 (`subnet_cmds` hook)

Description and examples: See Section [15.4.6.8](#)

Command syntax:



```
{
  "command": "subnet6-update",
  "arguments": {
    "subnet6": [ {
      "id": 234,
      "subnet": "2001:db8:1::/64",
      ...
    } ]
  }
}
```

Response syntax:

```
{
  "result": 0,
  "text": "IPv6 subnet updated",
  "arguments": {
    "subnet6": [
      {
        "id": 234,
        "subnet": "2001:db8:1::/64"
      }
    ]
  }
}
```

Result is an integer representation of the status. Currently supported statuses are:

- 0 - success
- 1 - error
- 2 - unsupported
- 3 - empty (command was completed successfully, but no data was affected or returned)

A.130 version-get reference

version-get - The version-get command returns extended information about Kea version. The returned string is the same as if Kea would be run with -V command line option.

Supported by: **kea-dhcp4**, **kea-dhcp6**, **kea-dhcp-ddns**, **kea-ctrl-agent**

Availability: 1.2.0 (built-in)

Description and examples: See Section [17.3.13](#)

Command syntax:

```
{
  "command": "version-get"
}
```

Response syntax:

```
{
  "result": <integer>,
  "text": <string>
}
```

Result is an integer representation of the status. Currently supported statuses are:



- 0 - success
 - 1 - error
 - 2 - unsupported
 - 3 - empty (command was completed successfully, but no data was affected or returned)
-



Appendix B

Acknowledgments

Kea is an open source project designed, developed, and maintained by Internet Systems Consortium, Inc, a 501(c)3 non-profit organization. ISC is primarily funded by revenues from support subscriptions for our open source, and we encourage all professional users to consider this option. To learn more, see <https://www.isc.org/support/>.

If you would like to contribute to ISC to assist us in continuing to make quality open source software, please visit our donations page at <https://www.isc.org/donate/>.

We thank all the organizations and individuals who have helped to make Kea possible. **Comcast** and the Comcast Innovation Fund provided major support for the development of Kea's DHCPv4, DHCPv6, and DDNS modules. Mozilla funded initial work on the REST API via a MOSS award.

Kea was initially implemented as a collection of applications within the BIND 10 framework. We thank the founding sponsors of the BIND 10 project: **Afilias, IIS.SE, Nominet, SIDN, JPRS, CIRA**; and additional sponsors **AFNIC, CNNIC, CZ.NIC, DENIC eG, Google, RIPE NCC, Registro.br, .nz Registry Services, and Technical Center of Internet** .
