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 Kea servers directly . . . 4

3 Installation 5

3.1 Packages . . . 5

3.2 Install 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 . . . 6

3.4.3 Configure before the build . . . 7

3.4.4 Build . . . 7

3.4.5 Install . . . 7

3.5 Selecting the Configuration Backend . . . 8

3.6 DHCP Database Installation and Configuration . . . 8

3.6.1 Building with MySQL Support . . . 8

3.6.2 Create the MySQL Database and the Kea User . . . 9

3.6.3 Building with PostgreSQL support . . . 9

3.6.4 Create PostgreSQL Database and Kea User . . . 9

4 Kea configuration 11

4.1 BUNDY configuration backend . . . 11

4.2 JSON configuration backend . . . 11

4.2.1 JSON syntax . . . 11

4.2.2 Simplified Notation . . . 12

5 Managing Kea with keactrl 13

5.1 Overview 13

5.2 Command Line Options 13

5.3 The keactrl Configuration File 13

5.4 Commands 14

5.5 Overriding the Server Selection 15

6 The DHCPv4 Server 16

6.1 Starting and Stopping the DHCPv4 Server 16

6.2 DHCPv4 Server Configuration 16

6.2.1 Introduction 16

6.2.2 Lease Storage 18

6.2.2.1 Memfile - Basic Storage for Leases 18

6.2.2.2 Database Configuration 19

6.2.3 Interface selection 19

6.2.4 IPv4 Subnet Identifier 20

6.2.5 Configuration of IPv4 Address Pools 20

6.2.6 Standard DHCPv4 options 21

6.2.7 Custom DHCPv4 options 25

6.2.8 DHCPv4 Vendor Specific Options 26

6.2.9 Nested DHCPv4 Options (Custom Option Spaces) 27

6.2.10 Stateless Configuration of DHCPv4 clients 29

6.2.11 Client Classification in DHCPv4 30

6.2.11.1 Limiting Access to IPv4 Subnet to Certain Classes 30

6.2.12 Configuring DHCPv4 for DDNS 30

6.2.12.1 DHCP-DDNS Server Connectivity 31

6.2.12.2 When Does the kea-dhcp4 Server Generate DDNS Requests? 32

6.2.12.3 kea-dhcp4 name generation for DDNS update requests 33

6.2.13 Next Server (siaddr) 34

6.2.14 Echoing Client-ID (RFC 6842) 34

6.3 Server Identifier in DHCPv4 35

6.4 How the DHCPv4 Server Selects a Subnet for the Client 35

6.4.1 Using a Specific Relay Agent for a Subnet 35

6.4.2 Segregating IPv4 Clients in a Cable Network 36

6.5 Supported DHCP Standards 36

6.6 DHCPv4 Server Limitations 37

7 The DHCPv6 Server 38

7.1 Starting and Stopping the DHCPv6 Server 38

7.2 DHCPv6 Server Configuration 38

7.2.1 Introduction 38

7.2.2 Lease Storage 40

7.2.2.1 Memfile - Basic Storage for Leases 40

7.2.2.2 Database Configuration 41

7.2.3 Interface selection 41

7.2.4 IPv6 Subnet Identifier 42

7.2.5 Unicast traffic support 42

7.2.6 Subnet and Address Pool 43

7.2.7 Subnet and Prefix Delegation Pools 44

7.2.8 Standard DHCPv6 options 44

7.2.9 Custom DHCPv6 options 46

7.2.10 DHCPv6 vendor specific options 48

7.2.11 Nested DHCPv6 options (custom option spaces) 49

7.2.12 IPv6 Subnet Selection 50

7.2.13 DHCPv6 Relays 51

7.2.14 Client Classification in DHCPv6 52

7.2.15 Limiting access to IPv6 subnet to certain classes 52

7.2.16 Configuring DHCPv6 for DDNS 53

7.2.16.1 DHCP-DDNS Server Connectivity 53

7.2.16.2 When does kea-dhcp6 generate DDNS request 54

7.2.16.3 kea-dhcp6 name generation for DDNS update requests 55

7.3 Server Identifier in DHCPv6 56

7.4 Using specific relay agent for a subnet 56

7.5 Segregating IPv6 clients in a cable network 57

7.6 Supported DHCPv6 Standards 58

7.7 DHCPv6 Server Limitations 58

8 The DHCP-DDNS Server 59

8.1 Starting and Stopping the DHCP-DDNS Server 59

8.2 Configuring the DHCP-DDNS Server 60

8.2.1 Global Server Parameters 60

8.2.2 TSIG Key List 61

8.2.3 Forward DDNS 62

8.2.3.1 Adding Forward DDNS Domains 62

8.2.3.1.1 Adding Forward DNS Servers 63

8.2.4 Reverse DDNS 64

8.2.4.1 Adding Reverse DDNS Domains 64

8.2.4.1.1 Adding Reverse DNS Servers 65

8.2.5 Example DHCP-DDNS Server Configuration 65

8.3 DHCP-DDNS Server Limitations 67

9 libdhcp++ library 68

9.1 Interface detection and Socket handling . . . 68

10 Logging 69

10.1 Logging configuration . . . 69

10.1.1 Loggers . . . 69

10.1.1.1 name (string) . . . 69

10.1.1.2 severity (string) . . . 70

10.1.1.3 output_options (list) . . . 70

10.1.1.4 debuglevel (integer) . . . 70

10.1.1.5 additive (true or false) . . . 70

10.1.2 Output Options . . . 71

10.1.2.1 destination (string) . . . 71

10.1.2.2 output (string) . . . 71

10.1.2.2.1 maxsize (integer) . . . 71

10.1.2.2.2 maxver (integer) . . . 71

10.1.3 Example Logger configurations . . . 71

10.2 Logging Message Format . . . 72

11 Acknowledgements 73

List of Tables

6.1 List of standard DHCPv4 options 23
6.2 List of standard DHCPv4 options (continued) 24
6.3 List of standard DHCP option types 24
6.4 Default FQDN Flag Behavior 32

7.1 List of standard DHCPv6 options 46
7.2 Default FQDN Flag Behavior 54

8.1 Our example network 65
8.2 Forward DDNS Domains Needed 66
8.3 Reverse DDNS Domains Needed 66

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 0.9. 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 at http://kea.isc.org/docs.

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.

Kea was initially developed as a part of the BIND 10 framework (http://bind10.isc.org). In early 2014, ISC made the decision to discontinue active development of BIND 10 and continue development of Kea as standalone DHCP software.

This guide covers Kea version 0.9.

1.1 Supported Platforms

Kea is officially supported on RedHat Enterprise Linux, CentOS, Fedora and FreeBSD systems. It is also likely to work on many other platforms: builds have been tested on (in no particular order) Debian GNU/Linux 6 and unstable, Ubuntu 9.10, NetBSD 5, Solaris 10 and 11, FreeBSD 7 and 8, CentOS Linux 5.3, MacOS 10.6 and 10.7, and OpenBSD 5.1. Non supported systems (especially non-Linux) are likely to have issues with directly connected DHCPv4 clients.

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

1.2 Required Software at Run-time

Running Kea uses various extra software 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 crypto libraries: Botan and OpenSSL. Only one of them is required to be installed during compilation. Kea uses the Botan crypto library for C++ (http://botan.randombit.net/), version 1.8 or later. As an alternative to Botan, Kea can use the OpenSSL crypto library (http://www.openssl.org/). It requires a version with SHA-2 support.
- Kea uses the log4cplus C++ logging library (http://log4cplus.sourceforge.net/). It requires at least log4cplus version 1.0.3.
- 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.

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** — DHCPv4 server process. This process responds to DHCPv4 queries from clients.
- **kea-dhcp6** — DHCPv6 server process. This process responds to DHCPv6 queries from clients.
- **kea-dhcp-ddns** — DHCP-DDNS process. This process acts as an intermediary between the DHCP servers and DNS server. It receives name update requests from the DHCP servers and sends DNS Update messages to the DNS servers.
- **perfdhcp** — DHCP benchmarking tool which simulates multiple clients to test both DHCPv4 and DHCPv6 servers 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 quickly covers the standard steps for installing and deploying Kea. For further details, full customizations, and troubleshooting, see the respective chapters in the Kea guide.

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 Kea source tarball from ISC.org downloads page or ISC ftp server.
3. Extract the tarball. For example:

   ```
   $ tar xvf kea-0.9.tar.gz
   ```

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

   ```
   $ cd kea-0.9
   $ ./configure [your extra parameters]
   ```

5. Build it:

   ```
   $ make
   ```

6. Install it (by default the installation prefix is `/usr/local/`, so you likely need root privileges for that step):

   ```
   # make install
   ```

7. Edit configuration file which is by default installed in `[kea-install-dir]/etc/kea/kea.conf` and contains configuration for all Kea services. Configuration choices for DHCPv4 and DHCPv6 services are described in Section 6.2 and Section 7.2, respectively.
8. In order to start the DHCPv4 server in background, run the following command (as root):

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

   Or run the following command to start DHCPv6 server instead:

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

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

   ```
   $ keactrl start
   ```

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

   ```
   # keactrl status
   ```

   If the server status is "inactive" may indicate a configuration error. Please check a log file (by default located in `[kea-install-` `/var/kea/kea.log`) for the details of the error.

10. If the server has been started sucessfully, 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 server(s):

    ```
    # keactrl stop
    ```

For more system specific installation procedures, you may want to visit System specific notes, available on Kea homepage.

The details of **keactrl** script usage can be found in Chapter 5.

2.2 Running Kea servers directly

Kea servers can be started directly (without a need to use **keactrl**). To start DHCPv4 server run the following commmand:

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

And, 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 build-only prerequisites and do not need to *make* the software.

FreeBSD ports, NetBSD pkgsrc, and Debian *testing* package collections provide all the prerequisite packages.

3.2 Install Hierarchy

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

- `etc/kea/` — configuration files.
- `include/` — C++ development header files.
- `lib/` — 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/kea/` — server identification, lease databases, and log 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 build-time headers (http://www.boost.org/). At least Boost version 1.35 is required.
- Botan (at least version 1.8) or OpenSSL.
- log4cplus (at least version 1.0.3) development include headers.
- A C++ compiler and standard development headers. Kea builds have been tested with GCC g++ 3.4.3, 4.1.2, 4.1.3, 4.2.1, 4.3.2, and 4.4.1; Clang++ 2.8; and Sun C++ 5.10.
- The development tools "make".

Visit the user-contributed wiki at http://kea.isc.org/wiki/SystemSpecificNotes for system-specific installation tips.

3.4 Installation from Source

Kea is open source software written in C++. It is freely available in source code form from ISC as a downloadable tar file or via Kea Git code revision control service. (It may also be available in pre-compiled ready-to-use packages from operating system vendors.)

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 (2.59 or later). These may need to be installed.

The latest development code (together with temporary experiments and un-reviewed code) is available via the Kea code revision control system. This is powered by Git and all the Kea development is public. The leading development is done in the “master” branch.

The code can be checked out from `git://git.kea.isc.org/kea`:

```
$ git clone git://git.kea.isc.org/kea
```

The code checked out from the git repository doesn’t 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.

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-boost-include Define the path to find the Boost headers.

--with-botan-config To specific the path to the botan-config script to build with Botan for the crypto code.

--with-gtest Enable the building of the C++ Unit Tests using the Google Test framework. Optionally this can define the path to the gtest header files and library. (If the framework is not already installed on your system, it can be downloaded from `https://code.google.com/p/googletest`.)

--with-log4cplus Define the path to find the Log4cplus headers and libraries.

--with-openssl Replace Botan by OpenSSL for the crypto library. The default is to try to find a working Botan then OpenSSL only if Botan is not found.

--without-werror Disable the default use of the `-Werror` compiler flag so that compiler warnings do not result in build failures.

> **Note**
> For additional instructions concerning the building and installation of Kea for various databases, see Section 3.6. For additional instructions concerning configuration backends, see Section 3.5.

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-dhcp-pgsql=/usr/local/bin/pg_config \
      --prefix=/opt/kea
```

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

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 Selecting the Configuration Backend

Kea 0.9 introduces configuration backends that are switchable during compilation phase. The backend is chosen using the --with-kea-config switch when running the configure script. It currently supports two values: BUNDY and JSON. JSON is the default.

BUNDY BUNDY means that Kea is linked with the Bundy configuration backend that connects to the Bundy framework and in general works exactly the same as Kea 0.8 and earlier BIND10 versions. The benefits of that backend are uniform integration with the Bundy framework, easy on-line reconfiguration using bindctl, available RESTful API. On the other hand, it requires the whole heavy Bundy framework that requires Python3 to be present. That backend is likely to go away with the release of Kea 1.0.

JSON JSON is a new default configuration backend that causes Kea to read JSON configuration file from disk. It does not require any framework and thus is considered more lightweight. It will allow dynamic on-line reconfiguration, but will lack remote capabilities (i.e. no RESTful API).

3.6 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 three database backends: MySQL, PostgreSQL and Memfile. To limit external dependencies, both MySQL and PostgreSQL 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 MySQL and/or PostgreSQL and the creation of the lease database.

3.6.1 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), do:

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

Or specify the location of the MySQL configuration program "mysql_config" if MySQL was not installed in the default location:

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

3.6.2 Create the MySQL Database and the Kea User

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

1. Log into MySQL as "root":

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

2. Create the database:

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

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

3. Create the database tables by running the dhcpdb_create.mysql script supplied as part of Kea:

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

4. 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';
```

5. Exit MySQL:

```
mysql> quit
Bye
$
```

3.6.3 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), do:

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

Or specify the location of the PostgreSQL configuration program "pg_config" if PostgreSQL was not installed in the default location:

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

3.6.4 Create PostgreSQL Database and Kea User

The next task is to create both the lease 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=#
postgres=# GRANT ALL PRIVILEGES ON DATABASE database-name TO user-name;
GRANT
postgres=#
```

4. Exit PostgreSQL:

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

5. Create the database tables using the new user's credentials and the dhcpdb_create.pgsql script supplied with Kea. After entering the following command, you will be prompted for the new user's password. When the command completes you will be returned to the shell prompt. You should see output similar to following:

```
$ psql -d database-name -U user-name -f path-to-kea/share/kea/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
$
```

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 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 following should be sufficient to provide password-authenticated access to Kea's database:

```
local   database-name    user-name                                 password
host    database-name    user-name          127.0.0.1/32           password
host    database-name    user-name          ::1/128                password
```

Please consult your PostgreSQL user manual before making these changes as they may expose your other databases that you run on the same system.

Chapter 4

Kea configuration

Depending on configuration backend chosen (see Section 3.5), configuration mechanisms are different. The following sections describe details of the different configuration backends. Note that only one configuration backend can be used and its selection is made when the configure script is run.

4.1 BUNDY configuration backend

This legacy configuration backend allows Kea to use the former BIND 10 framework. That framework and this Kea configuration backend is no longer supported by ISC. It is currently developed as part of the Bundy project (see Bundy homepage). See the Bundy project documentation regarding configuration.

4.2 JSON configuration backend

JSON is the default configuration backend. It assumes that the servers are started from the command line (either directly or using a script, e.g. `keactrl`). The JSON backend uses certain signals to influence Kea. The configuration file is specified upon startup using -c parameter.

4.2.1 JSON syntax

Configuration files for DHCPv4, DHCPv6 and DDNS modules are defined in an extended JSON format. Basic JSON is defined in RFC 4627. Kea components use a slightly modified JSON, in that they allow shell-style comments in the file: lines with the hash (#) character in the first column are comment lines and are ignored.

The configuration file consists of a single object (often colloquially called a map) started with a curly bracket. It comprises the "Dhcp4", "Dhcp6", "DhcpDdns" and/or "Logging" objects. It is possible to define additional elements, but they will be ignored. For example, it is possible to define Dhcp4, Dhcp6 and Logging elements in a single configuration file that can be used to start both the DHCPv4 and DHCPv6 components. When starting, the DHCPv4 component will use Dhcp4 object to configure itself and the Logging object to configure logging parameters; it will ignore the Dhcp6 object.

For example, a very simple configuration for both DHCPv4 and DHCPv6 could look like this:

```
# The whole configuration starts here.
{

# DHCPv4 specific configuration starts here.
"Dhcp4": {
    "interfaces": [ "eth0" ],
    "valid-lifetime": 4000,
    "renew-timer": 1000,
    "rebind-timer": 2000,
    "subnet4": [{
       "pool": "192.0.2.1-192.0.2.200",
       "subnet": "192.0.2.0/24"
    }],
    ...
},
# DHCPv4 specific configuration ends here.

# DHCPv6 specific configuration starts here.
"Dhcp6": {
    "interfaces": [ "eth1" ],
    "preferred-lifetime": 3000,
    "valid-lifetime": 4000,
    "renew-timer": 1000,
    "rebind-timer": 2000,
    "subnet6": [{
       "pool": "2001:db8::/80",
       "subnet": "2001:db8::/64"
    }],
    ...
},
# DHCPv6 specific configuration ends here.

# Logger parameters (that could be shared among several components) start here.
# This section is used by both the DHCPv4 and DHCPv6 servers.
"Logging": {
   "loggers": [{
        "name": "*",
        "severity": "DEBUG"
    }],
    ...
}
# Logger parameters end here.

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

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

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 remaining elements (the global map that holds Dhcp6, Logging and possibly DhcpDdns) are present, but they are omitted for clarity. Usually, locations where extra parameters may appear are denoted with an ellipsis.

4.2.2 Simplified Notation

It is sometimes convenient to refer to 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 referred by a number in square brackets (with numbering starting at zero). For example, in the above configuration the valid-lifetime in Dhcp6 component can be referred to as Dhcp6/valid-lifetime, the first interface for the DHCPv4 server as Dhcp4/interfaces[0] and the pool in the first subnet defined in the DHCPv6 configuration as Dhcp6/subnet6[0]/pool.

Chapter 5

Managing Kea with keactrl

5.1 Overview

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

5.2 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 5.4.

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.

5.3 The keactrl Configuration File

Depending on requirements, not all of the available servers need 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 servers.

# prefix holds the location where the Kea is installed.
prefix=/usr/local

# Location of Kea configuration file.
kea_config_file=${prefix}/etc/kea/kea.conf
```

```
# Location of Kea binaries.
exec_prefix=${prefix}
dhcp4_srv=${exec_prefix}/sbin/kea/kea-dhcp4
dhcp6_srv=${exec_prefix}/sbin/kea/kea-dhcp6
dhcp_ddns_srv=${exec_prefix}/sbin/kea/kea-dhcp-ddns

# Start DHCPv4 server?
dhcp4=yes

# Start DHCPv6 server?
dhcp6=yes

# Start DHCP DDNS server?
dhcp_ddns=yes

# Be verbose?
kea_verbose=no
```

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

By default, Kea servers managed by **keactrl** are located in `[kea-install-dir]/sbin`. This should work for most of the installations. If the default location needs to be altered for any reason, the paths specified with the *dhcp4_srv*, *dhcp6_srv* and *dhcp_ddns_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. Otherwise, the default logging level is 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.

5.4 Commands

The following commands are supported by **keactrl** to perform specific operations on the Kea servers:

- **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.

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.conf
INFO/keactrl: Starting kea-dhcp6 -c /usr/local/etc/kea/kea.conf
INFO/keactrl: Starting kea-dhcp-ddns -c /usr/local/etc/kea/kea.conf
```

The following command stops all servers:

```
$ keactrl stop
INFO/keactrl: Skip sending signal 15 to process kea-dhcp6: process is not running
```

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

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
```

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, typical output looking like:

```
$ keactrl status
DHCPv4 server: active
DHCPv6 server: inactive
DHCP DDNS: active
Kea configuration file: /usr/local/etc/kea/kea.conf
keactrl configuration file: /usr/local/etc/kea/keactrl.conf
```

5.5 Overriding the Server Selection

The optional **-s** switch allows the selection of the servers to which **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** server running:

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

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

```
$ 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.**
- **all** for all servers (default).

Chapter 6

The DHCPv4 Server

6.1 Starting and Stopping the DHCPv4 Server

It is recommended that the Kea DHCPv4 server be started and stopped using **keactrl** (described in Chapter 5). 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 a 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** ***`port`*** - specifies UDP port the server will listen on. This is only useful during testing, as the DHCPv4 server listening on ports other than default DHCPv4 ports will not be able to handle regular DHCPv4 queries.
- **-v** - prints out Kea version and exits.
- **-V** - prints out Kea extended version with additional parameters and exits.

When running in a console, the server can be shut down by pressing ctrl-c. It detects the key combination and shuts down gracefully.

On start-up, 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.

6.2 DHCPv4 Server Configuration

6.2.1 Introduction

This section explains how to configure the DHCPv4 server using the Kea configuration backend. (Kea configuration using any other backends is outside of scope of this document.) Before DHCPv4 is started, its configuration file has to be created. The basic configuration looks as follows:

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

# First we set up global values
    "interfaces": [ "eth0" ],
    "valid-lifetime": 4000,
    "renew-timer": 1000,
    "rebind-timer": 2000,

# Next we specify the type of lease database
    "lease-database": {
        "type": "memfile",
        "persist": true,
        "name": "/var/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 this line
}

}
```

The following paragraphs provide a brief overview of the parameters in the above example and 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 consists of one or more objects. In this specific example, we have only one object called Dhcp4. This is a simplified configuration, as usually there will be additional objects, like **Logging** or **DhcpDns**, but we omit them now for clarity. 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 general case, the order in which those parameters appear does not matter. There are two caveats here though. The first one is to remember that the configuration file must be a well formed JSON. That means that parameters for any given scope must be separate by a comma and there must not be a comma after the last parameter. When reordering configuration file, keep in mind that moving a parameter to or from the last position in a given scope may require moving the comma as well. The second caveat is that it is uncommon — although legal JSON — to repeat the same parameter multiple times. If that appears, 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 line defining **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. Had we wanted to listen on two interfaces, the line could look like this:

```
"interfaces": [ "eth0", "eth1" ],
```

As "**interfaces**" is not the last parameter in the configuration, a trailing comma is required.

A number of other parameters follow. **valid-lifetime** defines how long the addresses (leases) given out by the server are valid. If nothing changes, client that got the 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 that define T1 and T2 timers that govern when the client will begin renewal and rebind procedures. Note that **renew-timer** and **rebind-timer** are optional. If they are not specified the client will select values for T1 and T2 timers according to the RFC 2131.

The next couple 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 in-memory database and stores leases on disk in a CSV file. This is a very simple configuration. Usually, 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. Had there been more than one, they would be separated by commas. This scope is closed with a closing brace. As more parameters 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 that the server is expected to receive DHCP requests from. 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 is a structure and is opened and closed with braces. At minimum, a subnet definition has to have at least two parameters: **subnet** (that 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. Had more than one been 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"
    },
    {
        "pool": [ { "pool": "192.0.4.1 - 192.0.4.254" } ],
        "subnet": "192.0.4.0/24"
    }
]
```

After all parameters are specified, we have two contexts open: global and Dhcp4, hence we need two closing curly brackets to close them. In a real life configuration file there most likely would be additional components defined such as Logging or DhcpDdns, so the closing brace would be followed by a comma and another object definition.

Kea 0.9 does not have configuration syntax validation implemented yet. Such a feature is planned for the near future. For the time being, it is convenient to use on-line JSON validators and/or viewers to check whether the syntax is correct. One example of such a JSON validator is available at `http://jsonviewer.stack.hu/`.

6.2.2 Lease Storage

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

6.2.2.1 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 6.2.2.2 describes this option. In typical smaller deployments though, the server will use a CSV file rather than a database to store lease information. 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 following configuration sets the name of the lease file to `/tmp/kea-leases4.csv`:

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

The "persist" parameter controls whether the leases are written to disk. It is strongly recommended that this parameter is set to **true** at all times during the normal operation of the server. (Not writing leases to disk will mean 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.)

6.2.2.2 Database Configuration

Note

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.

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

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

Next, the name of the database is to hold the leases must be set: this is the name used when the lease database was created (see Section 3.6.2 or Section 3.6.4).

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

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

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

The usual state of affairs will be to have the database on the same machine as the DHCPv4 server. In this case, set the value to the empty string:

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

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.)

6.2.3 Interface selection

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

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

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

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

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

6.2.4 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 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 third is removed the last subnet will be assigned 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. It is planned to implement the mechanism to preserve auto-generated subnet ids in a future version of Kea. However, the only remedy for this issue at present is to manually specify a unique identifier for each subnet.

The following configuration will assign the arbitrary 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.

6.2.5 Configuration of IPv4 Address Pools

The essential role of DHCPv4 server is address assignment. The server has to 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 has decided 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 in this example.

Each **pool** is a structure that contains the parameters th describe a single pool. Currently there is only one parameter, **pool**, which gives the range of addresses in the pool. Additional parameters will be added in future releases of Kea.

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": [
        "pools": [
            { "pool": "192.0.2.10-192.0.2.20" },
            { "pool": "192.0.2.64/26" }
        ],
        ...
    ],
    ...
}
```

The number of pools is not limited, but for performance reasons it is recommended to use as few as possible. White space in pool definitions is ignored, so spaces before and after hyphen are optional. They can be used to improve readability.

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 should use a given pool, it will be able to allocate also first (typically network address) and the last (typically broadcast address) address from that pool. In the aforementioned example of pool 192.0.3.0/24, both 192.0.3.0 and 192.0.3.255 addresses may be assigned as well. This may be invalid in some network configurations. If you want to avoid this, please use the "min-max" notation.

6.2.6 Standard DHCPv4 options

One of the major features of the DHCPv4 server is 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 requested them. 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"
        },
        ...
    ]
}
```

The **name** parameter specifies the option name. For a complete list of currently supported names, see Table 6.1 below. The **code** parameter specifies the option code, which must match one of the values from that list. The next line specifies 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 6.2.9. 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 a 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"
        },
        ...
    ],
    ...
}
```

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 (Dhcp4/option-data), rather you should set only subnet-specific values (Dhcp4/subnet[X]/option-data[Y]).

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

Note

In future versions of Kea, it will not be necessary to specify the **code**, **space** and **csv-format** fields, as they will be set automatically.

The currently supported standard DHCPv4 options are listed in Table 6.1 and Table 6.2. The "Name" and "Code" are the values that should be used as a name in the option-data structures. "Type" designates the format of the data: the meanings of the various types is given in Table 6.3.

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, so allowing clients to obtain the addresses of multiple NTP servers.

The Section 6.2.7 describes the configuration syntax to create custom option definitions (formats). It is generally not allowed to create custom definitions for standard options, even if the definition being created matches the actual option format defined in the RFCs. There is an exception from this rule for standard options for which Kea does not provide a definition yet. In order to use such options, a server administrator must create a definition as described in Section 6.2.7 in the 'dhcp4' option space. This definition should match the option format described in the relevant RFC but configuration mechanism would allow any option format as it has no means to validate it at the moment.

Name	Code	Type	Array?
subnet-mask	1	ipv4-address	false
time-offset	2	int32	false
routers	3	ipv4-address	true
time-servers	4	ipv4-address	true
name-servers	5	ipv4-address	false
domain-name-servers	6	ipv4-address	true
log-servers	7	ipv4-address	true
cookie-servers	8	ipv4-address	true
lpr-servers	9	ipv4-address	true
impress-servers	10	ipv4-address	true
resource-location-servers	11	ipv4-address	true
host-name	12	string	false
boot-size	13	uint16	false
merit-dump	14	string	false
domain-name	15	fqdn	false
swap-server	16	ipv4-address	false
root-path	17	string	false
extensions-path	18	string	false
ip-forwarding	19	boolean	false
non-local-source-routing	20	boolean	false
policy-filter	21	ipv4-address	true
max-dgram-reassembly	22	uint16	false
default-ip-ttl	23	uint8	false
path-mtu-aging-timeout	24	uint32	false
path-mtu-plateau-table	25	uint16	true
interface-mtu	26	uint16	false
all-subnets-local	27	boolean	false
broadcast-address	28	ipv4-address	false
perform-mask-discovery	29	boolean	false
mask-supplier	30	boolean	false
router-discovery	31	boolean	false
router-solicitation-address	32	ipv4-address	false
static-routes	33	ipv4-address	true
trailer-encapsulation	34	boolean	false
arp-cache-timeout	35	uint32	false
ieee802-3-encapsulation	36	boolean	false
default-tcp-ttl	37	uint8	false
tcp-keepalive-internal	38	uint32	false
tcp-keepalive-garbage	39	boolean	false

Table 6.1: List of standard DHCPv4 options

Name	Code	Type	Array?
nis-domain	40	string	false
nis-servers	41	ipv4-address	true
ntp-servers	42	ipv4-address	true
vendor-encapsulated-options	43	empty	false
netbios-name-servers	44	ipv4-address	true
netbios-dd-server	45	ipv4-address	true
netbios-node-type	46	uint8	false
netbios-scope	47	string	false
font-servers	48	ipv4-address	true
x-display-manager	49	ipv4-address	true
dhcp-requested-address	50	ipv4-address	false
dhcp-option-overload	52	uint8	false
dhcp-message	56	string	false
dhcp-max-message-size	57	uint16	false
vendor-class-identifier	60	binary	false
nwip-domain-name	62	string	false
nwip-suboptions	63	binary	false
tftp-server-name	66	string	false
boot-file-name	67	string	false
user-class	77	binary	false
fqdn	81	record	false
dhcp-agent-options	82	empty	false
authenticate	90	binary	false
client-last-transaction-time	91	uint32	false
associated-ip	92	ipv4-address	true
subnet-selection	118	ipv4-address	false
domain-search	119	binary	false
vivco-suboptions	124	binary	false
vivso-suboptions	125	binary	false

Table 6.2: List of standard DHCPv4 options (continued)

Name	Meaning
binary	An arbitrary string of bytes, specified as a set of hexadecimal digits.
boolean	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)
record	Structured data that may comprise any types (except "record" and "empty")
string	Any text
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

Table 6.3: List of standard DHCP option types

6.2.7 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 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": [
        {
            "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 rather 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. This should be non-empty if the **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 set of comments define the format of the new option and do not set its values.

Note

In the current release the default values are not propagated to the parser when the new configuration is being set. Therefore, all parameters must be specified at all times, even if their values are left blank.

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. Assume we want to define a new option that will consist of an IPv4 address, followed by 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 those listed in Table 6.3.

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

Note
In general case, boolean values are specified as **true** or **false**, without quotes. Some specific boolean parameters may accept also **"true"**, **"false"**, **0**, **1**, **"0"** and **"1"**. Future Kea versions will accept all those values for all boolean parameters.

6.2.8 DHCPv4 Vendor Specific Options

Currently there are three option spaces defined: "dhcp4" (used by the DHCPv4 daemon) and "dhcp6" (for the DHCPv6 daemon); there is also "vendor-encapsulated-options-space", which is empty by default, but options can be defined in it. Those options are called vendor-specific information options. The following examples show how to define an option "foo" with code 1 that consists of 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",
            "encapsulates": ""
        }
    ],
    ...
}
```

(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 set up a dummy value for "vendor-encapsulated-options", the option that conveys our sub-option "foo". This is required else the option will not be included in messages sent to the client.

```
"Dhcp4": {
    "option-data": [
        {
            "name": "vendor-encapsulated-options"
            "space": "dhcp4",
            "code": 43,
            "csv-format": false,
            "data": ""
        }
    ],
    ...
}
```

Note

With this version of Kea, the "vendor-encapsulated-options" option must be specified in the configuration although it has no configurable parameters. If it is not specified, the server will assume that it is not configured and will not send it to a client. In the future there will be no need to include this option in the configuration.

6.2.9 Nested DHCPv4 Options (Custom Option Spaces)

It is sometimes useful to define completely new option space. This is the case when user creates new option in the standard option space ("dhcp4 or "dhcp6") 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 creation of a new option space when defining sub-options for a standard option is not required, because it is created by default if the standard option is meant to convey any sub-options (see Section 6.2.8).

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",
            "space": "isc",
            "code": 1,
            "csv-format": true,
            "data": "192.0.2.3"
        },
        }
            "name": "subopt2",
            "space": "isc",
            "code": 2,
            "csv-format": true,
            "data": "Hello world"
        },
        {
            "name": "container",
            "space": "dhcp4",
            "code": 222,
            "csv-format": true,
            "data": ""
        }
    ],
    ...
}
```

Even though the "container" option does not carry any data except sub-options, the "data" field must be explicitly set to an empty value. This is required because in the current version of Kea, the default configuration values are not propagated to the configuration parsers: if the "data" is not set the parser will assume that this parameter is not specified and an error will be reported.

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 was required to carry an 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.

6.2.10 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. DNS servers' addresses. 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 the particular subnet defined in the server's configuration. The 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, 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.

6.2.11 Client Classification in DHCPv4

Note

The DHCPv4 server has been extended to support limited client classification. Although the current capability is modest, it is expected to be expanded in the future. However, it is envisaged that the majority of client classification extensions will be using hooks extensions.

In certain cases it is useful to differentiate between different types of clients and treat them differently. The process of doing classification is conducted in two steps. The first step is to assess incoming packet and assign it to zero or more classes. This classification is currently simple, but is expected to grow in capability soon. Currently the server checks whether incoming packet has vendor class identifier option (60). If it has, content of that option is prepended with "VENDOR_CLASS_" then 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".

It is envisaged that the client classification will be used for changing the behavior of almost any part of the DHCP message processing, including assigning leases from different pools, assigning different option (or different values of the same options) etc. For now, there are only two mechanisms that are taking advantage of client classification: specific processing for cable modems and subnet selection.

For clients that belong to the VENDOR_CLASS_docsis3.0 class, the siaddr field is set to the value of next-server (if specified in a subnet). If there is boot-file-name option specified, its value is also set in the file field in the DHCPv4 packet. For eRouter1.0 class, the siaddr is always set to 0.0.0.0. That capability is expected to be moved to external hook library that will be dedicated to cable modems.

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. There are two classes of devices: the cable modem itself, which should be handled a lease from subnet A and all other devices behind the modem that 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 6.2.11.1.

6.2.11.1 Limiting Access to IPv4 Subnet to Certain Classes

In certain cases it beneficial to restrict access to certain subnets only to clients that belong to a given subnet. For details on client classes, see Section 6.2.11. This is an extension of a previous example from Section 6.2.5. 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 VENDOR_CLASS_docsis3.0 are allowed to use this subnet. 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" } ],
            "client-class": "VENDOR_CLASS_docsis3.0"
        }
    ],
    ...
}
```

Care should be taken with client classification as it is easy to prevent clients that do not meet class criteria to be denied any service altogether.

6.2.12 Configuring DHCPv4 for DDNS

As mentioned earlier, kea-dhcp4 can be configured to generate requests to the DHCP-DDNS server to update DNS entries. These requests are known as NameChangeRequests 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 FQDN, lease address, and DHCID

The parameters for controlling the generation of NCRs for submission to the DHCP-DDNS server are contained in the **dhcp-ddns** section of the kea-dhcp4 server configuration. The default values for this section are as follows:

```
"Dhcp4": {
    "dhcp-ddns": {
        "enable-updates": true,
        "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": false,
        "generated-prefix": "myhost",
        "qualifying-suffix": "example.com"
    },
    ...
}
```

6.2.12.1 DHCP-DDNS Server Connectivity

In order for NCRs to reach the DHCP-DDNS server, kea-dhcp4 must be able to communicate with it. kea-dhcp4 uses the following configuration parameters to control how it communications with DHCP-DDNS:

- **enable-updates** - determines whether or not kea-dhcp4 will generate NCRs. By default, this value is false hence DDNS updates are disabled. To enable DDNS updates set this value to true:
- **server-ip** - IP address on which DHCP-DDNS 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 DHCP-DDNS listens for requests. The default value is 53001.
- **sender-ip** - IP address which kea-dhcp4 should use to send requests to the DHCP-DDNS server. The default value is blank which instructs kea-dhcp4 to select a suitable address.
- **sender-port** - port which kea-dhcp4 should use to send requests to the DHCP-DDNS server. The default value of 0 instructs kea-dhcp4 to select suitable port.
- **max-queue-size** - maximum number of requests allowed to queue waiting to be sent to the DHCP-DDNS server. 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 intention is 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-format** - socket protocol use when sending requests to the DHCP-DDNS server. Currently only UDP is supported. TCP may be available in an upcoming release.
- **ncr-protocol** - packet format to use when sending requests to the DHCP-DDNS server. Currently only JSON format is supported. Other formats may be available in future releases.

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

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

6.2.12.2 When Does the kea-dhcp4 Server Generate DDNS Requests?

kea-dhcp4 follows the behavior prescribed for DHCP servers in RFC 4702. It is important to keep in mind that kea-dhcp4 provides the initial decision making of when and what to update and forwards that information to the DHCP-DDNS server in the form of NCRs. Carrying out the actual DNS updates and dealing with such things as conflict resolution are within the purview of the DHCP-DDNS server itself (Chapter 8). 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 DHCP REQUEST
2. An existing lease is renewed but the FQDN associated with it has changed.
3. An existing lease is released in response to a DHCP RELEASE

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 decision making involved when granting a new lease (the first case) is more involved and is discussed next.

When a new lease is granted, kea-dhcp4 will generate a DDNS update request if the DHCP REQUEST 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 6.4: 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 DHCP-DDNS 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 your 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 request to the DHCP-DDNS server 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, the following values should be set in your configuration:

```
"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.

6.2.12.3 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 DHCP REQUEST.

The rules for determining the FQDN option are as follows:

1. If configured to do, so ignore the REQUEST contents and generate a FQDN using a configurable prefix and suffix.
2. If the REQUEST contains the client FQDN option, the candidate name is taken from there, otherwise it is taken from the Host Name option. The candidate name may then be modified:
    (a) If the candidate name is a fully qualified domain name, use it.
    (b) 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.
    (c) If the candidate name is a empty, generate a FQDN using a configurable prefix and suffix.

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

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

The prefix used in the generation of a 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 a FQDN or when qualifying a partial name is specified by the **qualifying-suffix** parameter. There is no default value. 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 name of 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 lease address is 172.16.1.10 and assuming default values for **generated-prefix** and **qualifying-suffix**, the generated FQDN would be:

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

6.2.13 Next Server (siaddr)

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

```
"Dhcp4": {
    "next-server": "192.0.2.123",
    ...,
    "subnet4": {
        [
            "next-server": "192.0.2.234",
            ...
        ]
    }
}
```

6.2.14 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 MAC address or client-id; see RFC 6842. for details. That behavior has changed with the publication of RFC 6842. which updated RFC 2131. That update now states that the server must send client-id if client sent it. That is the default behaviour that Kea offers. However, in some cases older devices that do not support RFC 6842. may refuse to accept responses that include 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,
    ...
}
```

6.3 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.

Currently there is no mechanism to override the default server identifiers by an administrator. In the future, the configuration mechanism will be used to specify the custom server identifier.

6.4 How the DHCPv4 Server Selects a Subnet for the Client

The DHCPv4 server differentiates between the directly connected clients, clients trying to renew leases and clients sending their messages through relays. For the directly connected clients the server will check the configuration of 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 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 message is accepted through any interface. The renewing client sets ciaddr to the currently used IPv4 address. 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 subnets. See Section 6.4.1 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 subnet is selected for the client, depending on the classes that the client belongs to.

6.4.1 Using a Specific Relay Agent for a Subnet

The relay has to 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 that belongs to the subnet that the server will assign addresses from. In such typical case, the server is able to use 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 — 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 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 relay that uses 10.0.0.1 as its IPv4 address. The server will be able to select this subnet for any incoming packets that came from a relay that has an address in 192.0.2.0/24 subnet. It will also select 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-address": "10.0.0.1"
            },
            ...
        }
    ],
    ...
}
```

6.4.2 Segregating IPv4 Clients in a Cable Network

In certain cases, it is useful to mix relay address information, introduced in Section 6.4.1 with client classification, explained in Section 6.2.11.1. One specific example is cable network, where typically modems get addresses from a different subnet than all devices connected behind them.

Let's 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 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-address": "10.1.1.1"
            }
        },
        {
            "subnet": "192.0.2.0/24",
            "pools": [ { "pool": "192.0.2.10 - 192.0.2.20" } ],
            "relay": {
                "ip-address": "10.1.1.1"
            }
        }
    ],
    ...
}
```

6.5 Supported DHCP Standards

The following standards are currently supported:

- *Dynamic Host Configuration Protocol*, RFC 2131: Supported messages are DISCOVER (1), OFFER (2), REQUEST (3), RELEASE (7), INFORM (8), ACK (5), and NAK(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 6.2.14 for details.

6.6 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.

- Removal of a subnet during server reconfiguration may cause renumbering of auto-generated subnet identifiers, as described in section Section 6.2.4.
- Host reservation (static addresses) is not supported yet.
- Full featured client classification is not supported yet.
- 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 the system kernel. Currently, Kea can only parse the data link layer headers with a format adhering to 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 assigned address is unused. According to RFC 2131, the allocating server should verify that address is not used by sending ICMP echo request.
- Address duplication report (DECLINE) is not supported yet.
- The server doesn’t act upon expired leases. In particular, when a lease expires, the server doesn’t request the removal of the DNS records associated with it. Expired leases can be recycled.

Chapter 7

The DHCPv6 Server

7.1 Starting and Stopping the DHCPv6 Server

It is recommended that the Kea DHCPv4 server be started and stopped using **keactrl** (described in Chapter 5). 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.
- **-v** - specifies whether the server logging should be switched to verbose mode. In verbose mode, the logging severity and debuglevel specified in a 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** ***port*** - specifies UDP port the server will listen on. This is only useful during testing, as the DHCPv6 server listening on ports other than default DHCPv6 ports will not be able to handle regular DHCPv6 queries.
- **-v** - prints out Kea version and exits.
- **-V** - prints out Kea extended version with additional parameters and exits.

When running in a console, the server can be shut down by pressing ctrl-c. It detects the key combination and shuts down gracefully.

On start-up, 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.

7.2 DHCPv6 Server Configuration

7.2.1 Introduction

This section explains how to configure the DHCPv6 server using the Kea configuration backend. (Kea configuration using any other backends is outside of scope of this document.) Before DHCPv6 is started, its configuration file has to be created. The basic configuration looks as follows:

```
{
# DHCPv6 configuration starts on the next line
"Dhcp6": {

# First we set up global values
    "interfaces": [ "eth0" ],
    "renew-timer": 1000,
    "rebind-timer": 2000,
    "preferred-lifetime": 3000,
    "valid-lifetime": 4000,

# Next we specify the type of lease database
    "lease-database": {
        "type": "memfile",
        "persist": true,
        "name": "/var/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 and 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 consists of one or more objects. In this specific example, we have only one object called Dhcp6. This is a simplified configuration, as usually there will be additional objects, like **Logging** or **DhcpDns**, but we omit them now for clarity. 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 general case, the order in which those parameters appear does not matter. There are two caveats here though. The first one is to remember that the configuration file must be a well formed JSON. That means that parameters for any given scope must be separate by a comma and there must not be a comma after the last parameter. When reordering configuration file, keep in mind that moving a parameter to or from the last position in a given scope may require moving the comma as well. 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 line defining **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. Had we wanted to listen on two interfaces, the line could look like this:

```
"interfaces": [ "eth0", "eth1" ],
```

As "**interfaces**" is not the last parameter in the configuration, a trailing comma is required.

A number of other parameters follow. **valid-lifetime** defines how long the addresses (leases) given out by the server are valid. If nothing changes, client that got the 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 (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 renewal and rebind procedures.

The next couple 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 in-memory database and stores leases on disk in a CSV file. This is a very simple configuration. Usually, 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. Had there been more than one, they would be separated by commas. This scope is closed with a closing brace. As more parameters 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 that the server is expected to receive DHCP requests from. 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 is a structure and is opened and closed with braces. At minimum, a subnet definition has to have at least two parameters: **subnet** (that defines the whole subnet) and **pool** (which is a list of dynamically allocated pools that are governed by the DHCP server).

The example contains a single subnet. Had more than one been 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",
        "interface": "eth0"
    }
]
```

Note that indentation is optional and is used for aesthetic purposes only. In some cases in 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. In a real life configuration file there most likely would be additional components defined such as Logging or DhcpDdns, so the closing brace would be followed by a comma and another object definition.

Kea 0.9 does not have configuration syntax validation implemented yet. Such a feature is planned for the near future. For the time being, it is convenient to use on-line JSON validators and/or viewers to check whether the syntax is correct. One example of such a JSON validator is available at http://jsonviewer.stack.hu/.

7.2.2 Lease Storage

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

7.2.2.1 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 7.2.2.2 describes this option. In typical smaller deployments though, the server will use a CSV file rather than a database to store lease information. 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 following configuration sets the name of the lease file to `/tmp/kea-leases6.csv`:

```
"Dhcp6": {
    "lease-database": {
        "type": "memfile",
        "persist": true,
        "name": "/tmp/kea-leases6.csv"
    }
    ...
}
```

The "persist" parameter controls whether the leases are written to disk. It is strongly recommended that this parameter is set to true at all times during the normal operation of the server. (Not writing leases to disk will mean 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.)

7.2.2.2 Database Configuration

Note
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.

Database configuration is controlled through the Dhcp6/lease-database parameters. The type of the database must be set to "mysql" or "postgresql", e.g.

```
"Dhcp6": { "lease-database": { "type": "mysql", ... }, ... }
```

Next, the name of the database is to hold the leases must be set: this is the name used when the lease database was created (see Section 3.6.2 or Section 3.6.4).

```
"Dhcp6": { "lease-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 (although it should be noted that this configuration may have a severe impact on server performance):

```
"Dhcp6": { "lease-database": { "host": remote-host-name", ... }, ... }
```

The usual state of affairs will be to have the database on the same machine as the DHCPv6 server. In this case, set the value to the empty string:

```
"Dhcp6": { "lease-database": { "host" : "", ... }, ... }
```

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.)

7.2.3 Interface selection

The DHCPv6 server has to be configured to listen on specific network interfaces. The simplest network interface configuration tells the server to listen on all available interfaces:

```
"Dhcp6": { "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": [ "eth1", "eth3" ], ... }
```

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

```
"Dhcp6": { "interfaces": [ "eth1", "eth3", "*" ], ... }
```

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

7.2.4 IPv6 Subnet Identifier

Subnet identifier is a unique number associated with a particular subnet. In principle, it is used to associate clients' leases with respective subnets. When the 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 4 subnets and 3rd is removed the last subnet will be assigned identifier that the 3rd subnet had before removal. As a result, the leases stored in the lease database for subnet 3 are now associated with the subnet 4, which may have unexpected consequences. In the future it is planned to implement the mechanism to preserve auto-generated subnet ids upon removal of one of the subnets. Currently, the only remedy for this issue is to manually specify the unique subnet identifier for each subnet.

The following configuration will assign the arbitrary 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.

7.2.5 Unicast traffic support

When DHCPv6 server starts up, 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 7.2.3). In some cases it is useful to configure a server to handle incoming traffic sent to the global unicast addresses as well. The most common reason for that is to have relays send their traffic to the server directly. To configure server to listen on specific unicast address, a notation to specify interfaces has been extended. Interface name can be optionally followed by a slash, followed by global unicast address that server should listen on. That will be done in addition to normal link-local binding + listening on ff02::1:2 address. The sample commands listed below show how to listen on 2001:db8::1 (a global address) configured on the eth1 interface.

```
"Dhcp6": {
    "interfaces": [ "eth1/2001:db8::1"],
    ...
}
```

When configuration gets committed, the server will start to listen on eth1 on link-local address, multicast group (ff02::1:2) and 2001:db8::1.

It is possible to mix interface names, wildcards and interface name/addresses on the Dhcp6/interface list. It is not possible 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 those addresses specified, without any additional checks. That approach is selected on purpose, so in the software can be used to communicate over uncommon addresses if the administrator desires so.

7.2.6 Subnet and Address Pool

The essential role of a DHCPv6 server is address assignment. For this, the server has to 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 has decided 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 pool 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 th describe a single pool. Currently there is only one parameter, **pool**, which gives the range of addresses in the pool. Additional parameters will be added in future releases of Kea.

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 be also be managed by the server. It could be written as 2001:db8:1:0:5:: to 2001:db8:1::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" }
        ],
        ...
    }
    ]
}
```

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 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 of 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 should use a given pool, it will be able to allocate also first (typically network address) address from that pool. For example for pool 2001:db8:2::/64 the 2001:db8:2:: address may be assigned as well. If you want to avoid this, use the "min-max" notation.

7.2.7 Subnet and Prefix Delegation Pools

Subnets may also be configured to delegate prefixes, as defined in RFC 3633. A subnet may have one or more prefix delegation pools. Each pool has a prefixed address, which is specified as a prefix and a prefix length, as well as a delegated prefix length. **delegated-len** must not be shorter (numerically greater) than **prefix-len**. If both **delegated-len** and **prefix-len** are equal, the server will be able to delegate only one prefix. A sample configuration is shown below:

```
"Dhcp6": {
    "subnet6": [
        {
            "subnet": "2001:d8b:1::/64",
            "pd-pools": [
                {
                    "prefix": "2001:db8:1::",
                    "prefix-len": 64,
                    "delegated-len": 96
                }
            ]
        }
    ],
    ...
}
```

7.2.8 Standard DHCPv6 options

One of the major features of DHCPv6 server is 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 requested them. The following example shows how to configure DNS servers, which is one of the most frequently used options. Numbers in the first column are added for easier reference and will not appear on screen. 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 first line creates new entry in option-data table. It contains information on all global options that the server is supposed to configure in all subnets. The second line specifies option name. For a complete list of currently supported names, see Table 7.1. The third line specifies option code, which must match one of the values from that list. Line beginning with **space** specifies 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 7.2.11. The fifth 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 a 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 string of hexadecimal numbers. 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": "2001 0DB8 0001 0000 0000 0000 0000 CAFE
                    2001 0DB8 0001 0000 0000 0000 0000 BABE"
        },
        ...
    ]
}
```

The value for the setting of the "data" element is split across two lines in this document for clarity: when entering the command, the whole string should be entered on the same line. Care should be taken to use proper encoding when using hexadecimal format as Kea's ability to validate data correctness in hexadecimal is limited.

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 (Dhcp6/option-data), rather you should set only subnet-specific values (Dhcp6/subnet[X]/option-data[Y]).

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

Note

In future versions of Kea, it will not be necessary to specify the **code**, **space** and **csv-format** fields, as they will be set automatically.

The currently supported standard DHCPv6 options are listed in Table 7.1. The "Name" and "Code" are the values that should be used as a name in the option-data structures. "Type" designates the format of the data: the meanings of the various types is given in Table 6.3.

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, so allowing clients to obtain the addresses of multiple DNS servers.

The Section 7.2.9 describes the configuration syntax to create custom option definitions (formats). It is generally not allowed to create custom definitions for standard options, even if the definition being created matches the actual option format defined in the RFCs. There is an exception from this rule for standard options for which Kea does not provide a definition yet. In order to use such options, a server administrator must create a definition as described in Section 7.2.9 in the 'dhcp6' option space. This definition should match the option format described in the relevant RFC but configuration mechanism would allow any option format as it has no means to validate it at the moment.

Name	Code	Type	Array?
preference	7	uint8	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	false
remote-id	37	record	false
subscriber-id	38	binary	false
client-fqdn	39	record	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	44	record	false
client-data	45	empty	false
clt-time	46	uint32	false
lq-relay-data	47	record	false
lq-client-link	48	ipv6-address	true

Table 7.1: List of standard DHCPv6 options

7.2.9 Custom DHCPv6 options

It is also possible to define options other than the standard ones. Assume that we want to define a new DHCPv6 option called "foo" which will have code 100 and 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 rather 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-fields" value should be non-empty if the "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 set of comments define the format of the new option and do not set its values.

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.

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": "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 those listed in Table 6.3.

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

> **Note**
>
> In general case, boolean values are specified as **true** or **false**, without quotes. Some specific boolean parameters may accept also **"true"**, **"false"**, **0**, **1**, **"0"** and **"1"**. Future Kea versions will accept all those values for all boolean parameters.

7.2.10 DHCPv6 vendor specific options

Currently there are three option spaces defined: dhcp4 (to be used in DHCPv4 daemon) and dhcp6 (for the DHCPv6 daemon); there is also vendor-opts-space, which is empty by default, but options can be defined in it. Those options are called vendor-specific information options. 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. This option 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-encapsulated-options-space",
            "type": "record",
            "array": false,
            "record-types": "ipv6-address, uint16, string",
            "encapsulates": ""
        }
    ],
    ...
}
```

(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-encapsulated-options-space",
            "code": 1,
            "csv-format": true,
            "data": "2001:db8:1::10, 123, Hello World"
        },
        ...
    ],
    ...
}
```

We should also define values for the vendor-opts, that will convey our option foo.

```
"Dhcp6": {
    "option-data": [
        ...,
        {
            "name": "vendor-encapsulated-options"
            "space": "dhcp6",
            "code": 17,
            "csv-format": true,
            "data": "12345"
        }
    ],
    ...
}
```

7.2.11 Nested DHCPv6 options (custom option spaces)

It is sometimes useful to define completely new option spaces. This is useful if the user wants his new option to convey sub-options that use 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 it is not required to create new option space when defining sub-options for a standard option because it is by default created if the standard option is meant to convey any sub-options (see Section 7.2.10).

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 isc 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 imposes that this option does not carry any data other than sub-options.

Finally, we can set values for the new options:

```
"Dhcp6": {
    "option-data": [
        {
            "name": "subopt1",
            "space": "isc",
            "code": 1,
            "csv-format": true,
            "data": "2001:db8::abcd"
        },
        }
            "name": "subopt2",
            "space": "isc",
            "code": 2,
            "csv-format": true,
            "data": "Hello world"
        },
        {
            "name": "container",
            "space": "dhcp6",
            "code": 102,
            "csv-format": true,
            "data": ""
        }
    ],
    ...
}
```

Even though the "container" option does not carry any data except sub-options, the "data" field must be explicitly set to an empty value. This is required because in the current version of Kea, the default configuration values are not propagated to the configuration parsers: if the "data" is not set the parser will assume that this parameter is not specified and an error will be reported.

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 was required to carry an 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.

7.2.12 IPv6 Subnet Selection

The DHCPv6 server may receive requests from local (connected to the same subnet as the server) and remote (connecting via relays) clients. As server may have many subnet configurations defined, it must select appropriate subnet for a given request.

The server can not assume which of configured subnets are local. It is possible in IPv4, where there is reasonable expectation that the server will have a (global) IPv4 address configured on the interface, and can use that information to detect whether a subnet is local or not. That assumption is not true in IPv6, as the DHCPv6 must be able to operate with having link-local addresses only. Therefore an optional "interface" parameter is available within a subnet definition to designate that a given subnet is local, i.e. reachable directly over 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"
        }
    ],
    ...
}
```

7.2.13 DHCPv6 Relays

A DHCPv6 server with multiple subnets defined must select the appropriate subnet when it receives a request from 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 if 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|ipv6|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. In order 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 both reachable locally (direct traffic) and via relays (remote traffic). Specifying both is a configuration error and the DHCPv6 server will refuse such a configuration.

To specify interface-id with value "vlan123", the following commands can be used:

```
"Dhcp6": {
    "subnet6": [
        {
            "subnet": "2001:db8:beef::/48",
            "pools": [
                 {
                     "pool": "2001:db8:beef::/48"
                 }
             ],
            "interface-id": "vlan123"
        }
    ],
    ...
}
```

7.2.14 Client Classification in DHCPv6

Note

DHCPv6 server has been extended to support limited client classification. Although the current capability is modest, it is expected to be expanded in the future. It is envisaged that the majority of client classification extensions will be using hooks extensions.

In certain cases it is useful to differentiate between different types of clients and treat them differently. The process of doing classification is conducted in two steps. The first step is to assess incoming packet and assign it to zero or more classes. This classification is currently simple, but is expected to grow in capability soon. Currently the server checks whether incoming packet has vendor class option (16). If it has, content of that option is prepended with "VENDOR_CLASS_" 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".

It is envisaged that the client classification will be used for changing behavior of almost any part of the DHCP engine processing, including assigning leases from different pools, assigning different option (or different values of the same options) etc. For now, there is only one mechanism that is taking advantage of client classification: subnet selection.

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 are cable networks. There are two classes of devices: cable modem itself, which should be handled a lease from subnet A and all other devices behind modems that 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 7.2.15.

7.2.15 Limiting access to IPv6 subnet to certain classes

In certain cases it beneficial to restrict access to certain subnets only to clients that belong to a given subnet. For details on client classes, see Section 7.2.14. This is an extension of a previous example from Section 7.2.6. Let's assume that the server is connected to a network segment that uses the 2001:db8:1::/64 prefix. The Administrator of that network has decided that addresses from range 2001:db8:1::1 to 2001:db8:1::ffff are going to be managed by the Dhcp6 server. Only clients belonging to the eRouter1.0 client class are allowed to use that pool. Such a configuration can be achieved in the following way:

```
"Dhcp6": {
    "subnet6": [
        {
            "subnet": "2001:db8:1::/64",
            "pools": [
                 {
                     "pool": "2001:db8:1::-2001:db8:1::ffff"
                 }
             ],
            "client-class": "VENDOR_CLASS_eRouter1.0"
        }
    ],
    ...
}
```

Care should be taken with client classification as it is easy to prevent clients that do not meet class criteria to be denied any service altogether.

7.2.16 Configuring DHCPv6 for DDNS

As mentioned earlier, kea-dhcp6 can be configured to generate requests to the DHCP-DDNS server (referred to here as the "D2" server) to update DNS entries. These requests are known as NameChangeRequests 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 FQDN, lease address, and DHCID

The parameters controlling the generation of NCRs for submission to D2 are contained in the "dhcp-ddns" section of kea-dhcp6 configuration. The default values for this section appears as follows:

```
"Dhcp6": {
    "dhcp-ddns": {
        "enable-updates": true,
        "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": false,
        "generated-prefix": "myhost",
        "qualifying-suffix": "example.com"
    },
    ...
}
```

7.2.16.1 DHCP-DDNS Server Connectivity

In order 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 how it communications with D2:

- **enable-updates** - determines whether or not kea-dhcp6 will generate NCRs. If missing, this value is assumed to be false hence 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 should use 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 should use to send requests to D2. The default value of 0 instructs kea-dhcp6 to select 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 allow kea-dhcp6 to continue lease operations. The default value is 1024.
- **ncr-format** - Socket protocol use when sending requests to D2. Currently only UDP is supported. TCP may be available in an upcoming release.

- **ncr-protocol** - Packet format to use when sending requests to D2. Currently only JSON format is supported. Other formats may be available in future releases.

By default, D2 is assumed to 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 altered accordingly. For example, if D2 has been configured to listen on 2001:db8::5 port 900, the following commands would be required:

```
"Dhcp6": {
    "dhcp-ddns": {
        "server-ip": "2001:db8::5",
        "server-port": 900,
        ...
    },
    ...
}
```

7.2.16.2 When does kea-dhcp6 generate DDNS request

kea-dhcp6 follows the behavior prescribed for DHCP servers in RFC 4704. It is important to keep in mind that kea-dhcp6 provides the initial decision making 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 the purview of D2 (Chapter 8).

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. Support for multiple address mappings may be provided in a future release.

In general, kea-dhcp6 will generate DDNS update requests when:

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

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 decision making involved when granting a new lease is more involved and is discussed next.

kea-dhcp6 will generate a DDNS update request only if the DHCP REQUEST 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:

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 7.2: 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-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, issue the following commands:

```
"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 request to D2 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,
        ...
    },
    ...
}
```

7.2.16.3 kea-dhcp6 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-dhcp6 can be configured to supply a portion or all of that name based upon what it receives from the client in the DHCP REQUEST.

The rules for determining the FQDN option are as follows:

1. If configured to do so ignore the REQUEST contents and generate a FQDN using a configurable prefix and suffix.
2. Otherwise, using is the domain name value from the client FQDN option as the candidate name:
   (a) If the candidate name is a fully qualified domain name then use it.
   (b) 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.
   (c) If the candidate name is a empty then generate a FQDN using a configurable prefix and suffix.

To instruct kea-dhcp6 to always generate a FQDN, set the parameter "replace-client-name" to true:

```
"Dhcp6": {
    "dhcp-ddns": {
        "replace-client-name": true,
        ...
    },
    ...
}
```

The prefix used when generating a 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 a FQDN or when qualifying a partial name is specified by the **qualifying-suffix** parameter. There is no default value. 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 a name with the format:

[candidate-name].[qualifying-suffix].

where candidate-name is the partial name supplied in the REQUEST. For example, if FQDN domain name value was "some-computer" and assuming the default value for qualifying-suffix, the generated FQDN would be:

some-computer.example.com.

When generating a the entire name, kea-dhcp6 will construct name of 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 lease address is 3001:1::70E and assuming default values for generated-prefix and qualifying-suffix, the generated FQDN would be:

myhost-3001-1--70E.example.com.

7.3 Server Identifier in DHCPv6

The DHCPv6 protocol uses a "server identifier" (also known as a DUID) for clients to be able to discriminate between several servers present on the same link. There are several types of DUIDs defined, but RFC 3315 instructs servers to use DUID-LLT if possible. This format consists of a link-layer (MAC) address and a timestamp. When started for the first time, the DHCPv6 server will automatically generate such a DUID and store the chosen value to a file. That file is read by the server and the contained value used whenever the server is subsequently started.

It is unlikely that this parameter should ever need to be changed. However, if such a need arises, stop the server, edit the file and restart the server. (The file is named kea-dhcp6-serverid and by default is stored in the "var" subdirectory of the directory in which Kea is installed. This can be changed when Kea is built by using "--localstatedir" on the "configure" command line.) The file is a text file that contains double digit hexadecimal values separated by colons. This format is similar to typical MAC address format. Spaces are ignored. No extra characters are allowed in this file.

7.4 Using specific relay agent for a subnet

The relay has to have an interface connected to the link on which the clients are being configured. Typically the relay has a global IPv6 address configured on that interface that belongs to the subnet that the server will assign addresses from. In such typical case, the server is able to use IPv6 address inserted by the relay (in link-addr field in RELAY-FORW message) to select 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. 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 case, the DHCPv6 server needs additional information (like the value of interface-id option or IPv6 address inserted in the link-addr field in 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 relay that uses 3000::1 as its IPv6 address. The server will be able to select this subnet for any incoming packets that came from a relay that has an address in 2001:db8:1::/64 subnet. It will also select 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-address": "3000::1"
       }
        }
    ]
}
```

7.5 Segregating IPv6 clients in a cable network

In certain cases, it is useful to mix relay address information, introduced in Section 7.4 with client classification, explained in Section 7.2.15. One specific example is cable network, where typically modems get addresses from a different subnet than all devices connected behind them.

Let's 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 modems should get addresses from another subnet (2001:db8:1::/64). The CMTS that acts as a relay an 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-address": "3000::1"
            }
        },

        {
            "subnet": "2001:db8:1::/64",
            "pools": [
                 {
                     "pool": "2001:db8:1::1-2001:db8:1::ffff"
                 }
             ],
       "relay": {
           "ip-address": "3000::1"
       }
        }
    ]
}
```

7.6 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, 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) Client Fully Qualified Domain Name (FQDN) Option*, RFC 4704: Supported option is CLIENT_FQDN.

7.7 DHCPv6 Server Limitations

These are the current limitations and known problems with the DHCPv6 server software. Most of them are reflections of the early stage of development and should be treated as “not implemented yet”, rather than actual limitations.

- On-line configuration has some limitations. Adding new subnets or modifying existing ones work, as is removing the last subnet from the list. However, removing non-last (e.g. removing subnet 1,2 or 3 if there are 4 subnets configured) will cause issues. The problem is caused by simplistic subnet-id assignment. The subnets are always numbered, starting from 1. That subnet-id is then used in leases that are stored in the lease database. Removing non-last subnet will cause the configuration information to mismatch data in the lease database. It is possible to manually update subnet-id fields in MySQL or PostgreSQL database, but it is awkward and error prone process. A better reconfiguration support is planned.
- 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 3315 and RFC 3633 allow for multiple addresses or prefixes to be allocated for a single IA.
- Temporary addresses are not supported.
- Duplication report (DECLINE), stateless configuration (INFORMATION-REQUEST) and client reconfiguration (RECONFIGURE) are not yet supported.
- The server doesn’t act upon expired leases. In particular, when a lease expires, the server doesn’t request removal of the DNS records associated with it.

Chapter 8

The DHCP-DDNS Server

The DHCP-DDNS Server (kea-dhcp-ddns, known informally as D2) conducts the client side of the DDNS protocol (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 upon DHCP lease change events and then post these to D2. D2 attempts to match each such request to the appropriate DNS server(s) and carry out the necessary conversation with those servers to update the DNS data.

In order to match a request to 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 will compare the FQDN in the request against the name of each forward DDNS Domain. 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 will simply disregard the forward update portion of requests.

When conducting reverse domain matching, D2 constructs a reverse FQDN from the lease address in the request and compare 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 is possible to 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.

8.1 Starting and Stopping the DHCP-DDNS Server

kea-dhcp-ddns is the Kea DHCP-DDNS server and, due to the nature of DDNS, it is run alongside either the DHCPv4 or DHCPv6 components (or both). Like other parts of Kea, is a separate binary that can be run on its own or through **keactrl** (see Chapter 5). 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 a 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** - prints out Kea version and exits.

- **-V** - prints out Kea extended version with additional parameters and exits.

Upon start up the module will load its configuration and begin listening for NCRs based on that configuration.

8.2 Configuring the DHCP-DDNS Server

Before starting **kea-dhcp-ddns** module for the first time, a configuration file needs to be created. The following default configuration is a template that can be customised 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 as follows, each of which is described in its own section:

- **Global Server Parameters** - values which control connectivity and global server behavior
- **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

8.2.1 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** - The 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** - Packet format to use when sending requests to D2. Currently only JSON format is supported. Other formats may be available in future releases.
- **ncr_format** - Socket protocol to use when sending requests to D2. Currently only UDP is supported. TCP may be available in an upcoming release.

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. A future version of Kea will implement authentication to guard against such attacks.

Note

If the ip_address and port are changed, it will be necessary to change the corresponding values in the DHCP servers' "dhcp-ddns" configuration section.

8.2.2 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. So long as it 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). It 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.
- **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 BIND9 DNS server which requires dynamic updates to be secured with TSIG. Suppose further that the entry for the TSIG key in BIND9'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.

8.2.3 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.

8.2.3.1 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. You will need one forward DDNS Domain for each zone you wish to service. It may very well be that some or all of your zones are maintained by the same servers. 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.

The 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 is value used to compare 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 to use. 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 the it achieves success or the list is exhausted.

To create a new forward DDNS Domain, one must add a new domain element and set its parameters:

```
"DhcpDdns": {
    "forward_ddns": {
  "ddns_domains": [
      {
    "name": "other.example.com",
    "key_name": "",
    "dns_servers": [
    ]
      }
  ]
    }
}
```

It is permissible to add a domain without any servers. If that domain should be matched to a request, however, the request will fail. In order to make the domain useful though, we must add at least one DNS server to it.

8.2.3.1.1 Adding Forward DNS Servers

The section describes how to add DNS servers to a Forward DDNS Domain. Repeat them for as many servers as desired for a 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 value 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 its parameters. If for example the service is running at "172.88.99.10", then set it 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

As stated earlier, "hostname" is not yet supported so, the parameter "ip_address" must be set to the address of the DNS server.

8.2.4 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.

8.2.4.1 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. You will need one reverse DDNS Domain for each zone you wish to service. It may very well be that some or all of your zones are maintained by the same servers; even then, 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.

The 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 to support the IPv4 subnet 172.16.1, the name should be. "1.16.172.in-addr.arpa.". Similarly, to support an IPv6 subent of 2001:db8:1, the name should be "1.0.0.0.8.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, then 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 next one in the list and so on until the 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 permissible to add a domain without any servers. If that domain should be matched to a request, however, the request will fail. In order to make the domain useful though, we must add at least one DNS server to it.

8.2.4.1.1 Adding Reverse DNS Servers

The section describes how to add DNS servers to a Reverse DDNS Domain. Repeat them for as many servers as desired for a each domain.

Reverse DNS Server entries represents a 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 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
As stated earlier, "hostname" is not yet supported so, the parameter "ip_address" must be set to the address of the DNS server.

8.2.5 Example DHCP-DDNS Server Configuration

This section provides an example 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 8.1: Our example network

We need to construct three forward DDNS Domains: As discussed earlier, FQDN to domain matching is based on the longest 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 would be rejected.

The following example configuration specified the Forward DDNS Domains.

#	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 8.2: Forward DDNS Domains Needed

```
"DhcpDdns": {
    "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" }
    ]
      },

  ]
    }
}</b>
```

Similarly, we need to construct the three reverse DDNS Domains: An address of "192.0.2.150" will match the first domain, "2001:db8:1::10" will match the second domain, and "192.0.50.77" the third 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 8.3: Reverse DDNS Domains Needed

These Reverse DDNS Domains are specified as follows:

```
"DhcpDdns": {
    "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" }
    ]
      }
  ]
    }
}</b>
```

8.3 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 9

libdhcp++ library

libdhcp++ is a common library written in C++ that handles many DHCP-related tasks, including:

- DHCPv4 and DHCPv6 packets parsing, manipulation and assembly
- Option parsing, manipulation and assembly
- Network interface detection
- Socket operations such as creation, data transmission and reception and socket closing.

While this library is currently used by Kea, it is designed to be a portable, universal library, useful for any kind of DHCP-related software.

9.1 Interface detection and Socket handling

Both the DHCPv4 and DHCPv6 components share network interface detection routines. Interface detection is currently supported on Linux, all BSD family (FreeBSD, NetBSD, OpenBSD), Mac OS X and Solaris 11 systems.

DHCPv4 requires special raw socket processing to send and receive packets from hosts that do not have IPv4 address assigned yet. Support for this operation is implemented on Linux, FreeBSD, NetBSD and OpenBSD. It is likely that DHCPv4 component will not work in certain cases on other systems.

Chapter 10

Logging

10.1 Logging configuration

The logging system in Kea is configured through the Logging module. All modules will look at the configuration in Logging to see what should be logged and to where.

10.1.1 Loggers

Within Kea, a message is logged through a component called a "logger". Different parts of log messages through different loggers, and each logger can be configured independently of one another.

In the Logging module, you can specify the configuration for zero or more loggers; any that are not specified will take appropriate default values.

The three most important elements of a logger configuration are the `name` (the component that is generating the messages), the `severity` (what to log), and the `output_options` (where to log).

10.1.1.1 name (string)

Each logger in the system has a name, the name being that of the component using it to log messages. For instance, if you want to configure logging for the Dhcp4 module, you add an entry for a logger named “Dhcp4”. This configuration will then be used by the loggers in the Dhcp4 module, and all the libraries used by it (unless a library defines its own logger).

If you want to specify logging for one specific library within the module, you set the name to *`module.library`*. For example, the logger used by the nameserver address store component has the full name of “Dhcp4.dhcpsrv”. If there is no entry in Logging for a particular library, it will use the configuration given for the module.

To illustrate this, suppose you want the dhcpsrv library to log messages of severity DEBUG, and the rest of the Dhcp4 code to log messages of severity INFO. To achieve this you specify two loggers, one with the name “Dhcp4” and severity INFO, and one with the name “Dhcp4.dhcpsrv” with severity DEBUG. As there are no entries for other libraries, they will use the configuration for the module (“Dhcp4”), so giving the desired behavior.

One special case is that of a module name of “*” (asterisks), which is interpreted as *any* module. You can set global logging options by using this, including setting the logging configuration for a library that is used by multiple modules (e.g. “*.config” specifies the configuration library code in whatever module is using it).

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 “*” and “Dhcp4”, the Dhcp4 module — and all libraries it uses — will log messages according to the configuration in the second entry (“Dhcp4”). All other modules will use the configuration of the first entry (“*”).

One final note about the naming. When specifying the module name within a logger, use the name of the binary file, e.g. “kea-dhcp4” for the DHCPv4 module, “kea-dhcp6” for the DHCPv6 module, etc. When the message is logged, the message will include the name of the logger generating the message, but with the module name replaced by the name of the process implementing the module (so for example, a message generated by the “DHCPv4” logger will appear in the output with a logger name of “kea-dhcp4”).

Currently defined loggers are:

- kea-dhcp4.dhcp4
- kea-dhcp6.dhcp6
- kea-dhcp-ddns.dhcpddns
- kea-dhcp4.dhcpsrv
- kea-dhcp6.dhcpsrv

Additional loggers may be defined in the future.

10.1.1.2 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
- ERROR
- WARN
- INFO
- DEBUG

When the severity of a logger is set to one of these values, it will only log messages of that severity, and the severities above it. The severity may also be set to NONE, in which case all messages from that logger are inhibited.

10.1.1.3 output_options (list)

Each logger can have zero or more `output_options`. These specify where log messages are sent to. These are explained in detail below.

The other options for a logger are:

10.1.1.4 debuglevel (integer)

When a logger’s severity is set to DEBUG, this value specifies what 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.

10.1.1.5 additive (true or false)

If this is true, the `output_options` from the parent will be used. For example, if there are two loggers configured; “Dhcp4” and “Dhcp4.dhcpsrv”, and `additive` is true in the second, it will write the log messages not only to the destinations specified for “Dhcp4.dhcpsrv”, but also to the destinations as specified in the `output_options` in the logger named “Dhcp4”.

10.1.2 Output Options

The main settings for an output option are the `destination` and a value called `output`, the meaning of which depends on the destination that is set.

10.1.2.1 destination (string)

The destination is the type of output. It can be one of:

- console
- file
- syslog

10.1.2.2 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 that the logs should be written to.

The other options for `output_options` are:

10.1.2.2.1 maxsize (integer)

Only relevant when destination is file, this is maximum file size of output files in bytes. 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.)

If this is 0, no maximum file size is used.

> **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 or higher version of log4cplus is used to build Kea, it shouldn’t happen. Even for older versions it is normally expected to happen rarely unless the log messages are produced very frequently by multiple different processes.

10.1.2.2.2 maxver (integer)

Maximum number of old log files to keep around when rolling the output file. Only relevant when `output` is “file”.

10.1.3 Example Logger configurations

In this example we want to set the global logging to write to the console using standard output.

```
"Logging": {
    "loggers": [
        {
            "name": "kea-dhcp4",
            "output_options": [
                {
                    "output": "stdout"
                }
            ],
            "severity": "WARN"
        }
    ]
}
</b>
```

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 file be created.

```
"Logging": {
    "loggers": [
        {
            "name": "kea-dhcp6",
            "output_options": [
                {
                    "output": "/var/log/kea-debug.log",
                    "maxver": 8,
                    "maxsize": 204800,
                    "destination": "file"
                }
            ],
            "severity": "DEBUG",
            "debuglevel": 99
        }
   ]
}</b>
```

10.2 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.dhcpsrv/27456]
    DHCPSRV_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 comprises two components: the Kea process generating the message (in this case, **kea-dhcp4**) and the module within the program from which the message originated (which is the name of the common library used by DHCP server implementations).

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* (`http://kea.isc.org/docs/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.

Chapter 11

Acknowledgements

Kea is primarily designed, developed, and maintained by Internet Systems Consortium, Inc. It is an open source project and contributions are welcomed.

Support for the development of the DHCPv4, DHCPv6 and DHCP-DDNS components was provided by Comcast.

Kea was initially implemented as a collection of applications within the BIND 10 framework. Hence, Kea development would not be possible without the generous support of past BIND 10 project sponsors.

JPRS and CIRA were Patron Level BIND 10 sponsors.

AFNIC, CNNIC, CZ.NIC, DENIC eG, Google, RIPE NCC, Registro.br, .nz Registry Services, and Technical Center of Internet were past BIND 10 sponsors.

Afilias, IIS.SE, Nominet, and SIDN were founding sponsors of the BIND 10 project.