Kea Administrator Reference Manual

Contents

List of Tables

Abstract

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

This is the reference guide for Kea version 1.1.0. 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. 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 1.1.0.

1.1 Supported Platforms

Kea is officially supported on Red Hat Enterprise Linux, CentOS, Fedora and FreeBSD systems. It is also likely to work on many other platforms: Kea 1.1.0 builds have been tested on (in no particular order) Red Hat Enteprise Linux 6.4, Debian GNU/Linux 7, Ubuntu 12.04, Ubuntu 14.04, Ubuntu 16.04, Fedora Linux 19, Fedora 20, Fedora 22, CentOS Linux 7, NetBSD 6, FreeBSD 10.3, OpenBSD 5.7, OpenBSD 6.0, OS X 10.10, OS X 10.11.

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 cryptographic libraries: Botan and OpenSSL. Only one of them is required to be installed during compilation. If using Botan, Kea requires the Botan cryptographic library for C++ (http://botan.randombit.net/), version 1.8, 1.9 or 1.10. If OpenSSL is used, (http://www.openssl.org/), then Kea requires the OpenSSL C++ library version 1.0.*. Support for later versions of Botan and OpenSSL will be added in future releases of Kea.

- Kea uses the log4cplus C++ logging library (http://log4cplus.sourceforge.net/). It requires log4cplus version 1.0.3 or later.

- In order to store lease information in a MySQL database, Kea requires MySQL headers and libraries. This is an optional dependency in that Kea can be built without MySQL support.

- In order to store lease information in a PostgreSQL database, Kea requires PostgreSQL headers and libraries. This is an optional dependency in that Kea can be built without PostgreSQL support.

- In order to store lease information in a Cassandra database (CQL), Kea requires Cassandra headers and libraries. This is an optional dependency in that Kea can be built without Cassandra support.

1.3 Kea Software

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

- **keactrl** — Tool to start, stop, reconfigure, and report status for the Kea servers.

- **kea-dhcp4** — The DHCPv4 server process. This process responds to DHCPv4 queries from clients.

- **kea-dhcp6** — The DHCPv6 server process. This process responds to DHCPv6 queries from clients.

- **kea-dhcp-ddns** — The DHCP Dynamic DNS process. This process acts as an intermediary between the DHCP servers and DNS servers. It receives name update requests from the DHCP servers and sends DNS Update messages to the DNS servers.

- **kea-admin** — A useful tool for database backend maintenance (creating a new database, checking versions, upgrading etc.)

- **kea-lfc** — This process removes redundant information from the files used to provide persistent storage for the memfile data base backend. While it can be run standalone, it is normally run as and when required by the Kea DHCP servers.

- **perfdhcp** — A DHCP benchmarking tool which simulates multiple clients to test both DHCPv4 and DHCPv6 server performance.

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

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

Chapter 2

Quick Start

This section describes the basic steps needed to get Kea up and running. For further details, full customizations, and troubleshooting, see the respective chapters in the Kea 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 xvzf kea-1.1.0.tar.gz
   ```

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

   ```
   $ cd kea-1.1.0
   $ ./configure [your extra parameters]
   ```

5. Build it:

   ```
   $ make
   ```

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

   ```
   # make install
   ```

7. Edit the configuration file which by default is 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 7.2 and Section 8.2:w 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
   ```

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

10. If the server has been started successfully, test that it is responding to DHCP queries and that the client receives a configuration from the server; for example, use the ISC DHCP client.

11. Stop running the server(s):

    ```
    # keactrl stop
    ```

For instructions specific to your system, please read the system specific notes, available on the Kea web site.

The details of **keactrl** script usage can be found in Chapter 6.

2.2 Running the Kea Servers Directly

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

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

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

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

Chapter 3

Installation

3.1 Packages

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

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

3.2 Installation Hierarchy

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

- `bin/` — utility programs.
- `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 1.41 is required. When header-only Boost error code is not available or wanted, the Boost system library is required too.

- Botan (version 1.8, 1.9 or 1.10) or OpenSSL (versions 1.0.*).

- log4cplus (at least version 1.0.3) development include headers.

- A C++ compiler and standard development headers. Kea 1.1.0 builds have been tested with GCC g++ 4.2.1, 4.4.7, 4.6.3, 4.8.3, 4.8.4, 4.8.5, 5.4.0; Clang++ 3.4.1; and Apple Clang++ 703.0.31.

- The development tools automake, libtool, pkg-config.

- The MySQL client and the client development libraries, when using the --with-dhcp-mysql configuration flag to build the Kea MySQL database backend. In this case an instance of the MySQL server running locally or on a machine reachable over a network is required. Note that running the unit tests requires a local MySQL server.

- The PostgreSQL client and the client development libraries, when using the --with-dhcp-pgsql configuration flag to build the Kea PostgreSQL database backend. In this case an instance of the PostgreSQL server running locally or on some other machine, reachable over the network from the machine running Kea, is required. Note that running the unit tests requires a local PostgreSQL server.

- googletest (version 1.6 or later), when using the --with-gtest configuration option to build the unit tests.

- The documentation generation tools elinks, docbook-xsl, libxslt and Doxygen, if using the --enable-generate-docs configuration option to create the documentation.

Visit 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. A copy of the Kea source code repository is accessible from Github (https://github.com/isc-projects/kea). Kea 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 (v2.59 or later). These may need to be installed.

The latest development code is available on Github (see https://github.com/isc-projects/kea). The Kea source is public and development is done in the "master" branch.

The code can be checked out from https://github.com/isc-projects/kea.git:

```
$ git clone https://github.com/isc-projects/kea.git
```

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

Write access to the Kea repository is only granted to ISC staff. If you are a developer planning to contribute to Kea, please fork our Github repository and use the "pull request" mechanism to request integration of your code. Please consult https://help.github.com/articles/fork-a-repo/ for help on how to fork a Github repository. The Kea Developer's Guide contains more information about the process, as well as describing the requirements for contributed code to be accepted by ISC.

3.4.3 Configure Before the Build

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

```
$ ./configure
```

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

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

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

--with-botan-config
> Specify the path to the botan-config script to build with Botan for cryptographic functions.

--with-dhcp-mysql
> Build Kea with code to allow it to store leases (and access host reservations) in a MySQL database.

--with-dhcp-pgsql
> Build Kea with code to allow it to store leases (and access host reservations) in a PostgreSQL database.

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

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

--with-openssl
> Replace Botan by the OpenSSL the cryptographic library. By default **configure** searches for a valid Botan installation: if one is not found, it searches for OpenSSL.

Note
For instructions concerning the installation and configuration of database backends for Kea, see Section 3.6. For information concerning the 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 you have some problems with building Kea using the header-only Boost code or you'd like to use the Boost system library (assumed for the sake of this example to be located in /usr/pkg/lib):

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

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

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

3.4.4 Build

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

```
$ make
```

3.4.5 Install

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

```
$ make install
```

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

Note
The install step may require superuser privileges.

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

```
$ ldconfig
```

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

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

3.5 Selecting the Configuration Backend

Kea 0.9 introduced configuration backends that are switchable during the compilation phase. Only one backend, JSON, is currently supported.

JSON
    JSON is the new default configuration backend that allows Kea to read JSON configuration files from disk. It does not require any framework and thus is considered more lightweight. It will allow dynamic on-line reconfiguration, but lacks 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 four database backends: MySQL, PostgreSQL, Cassandra and Memfile. To limit external dependencies, MySQL, PostgreSQL and Cassandra support are disabled by default and only Memfile is available. Support for the optional external database backend must be explicitly included when Kea is built. This section covers the building of Kea with one of the optional backends and the creation of the lease database.

Note

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

3.6.1 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), the --with-dhcp-mysql switch should be specified:

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

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

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

See Section 4.3.2.1 for details regarding MySQL database configuration.

3.6.2 Building with PostgreSQL support

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

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

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

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

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

See Section 4.3.3.1 for details regarding PostgreSQL database configuration.

3.6.3 Building with CQL (Cassandra) support

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

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

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

As of June 2016, cpp-driver does not include cql_config script yet. Work is in progress to contribute such a script to the cpp-driver project but, until that is complete, intermediate steps that need to be conducted. A cql_config script is present in the tools/ directory of the Kea sources. Before using it, please edit cql_config_defines.sh in the same directory and change the environment variable CPP_DRIVER_PATH to point to the directory, where cpp-driver sources are located. (If the cpp-driver sources already provide cql_config script please use that rather than the version from Kea sources.)

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

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

Chapter 4

Kea Database Administration

4.1 Databases and Database Version Numbers

Kea supports storing leases and host reservations (i.e. static assignments of addresses, prefixes and options) in one of the several supported databases. As future versions of Kea are released, the structure of those databases will change. For example, Kea currently only stores lease information and host reservations. Future versions of Kea will store additional data such as subnet definitions: the database structure will need to be updated to accomdate the extra information.

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

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

4.2 The kea-admin Tool

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

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

- **lease-init** — Initializes a new lease database. This is useful during a new Kea installation. The database is initialized to the latest version supported by the version of the software being installed.

- **lease-version** — Reports the lease database version number. This is not necessarily equal to the Kea version number as each backend has its own versioning scheme.

- **lease-upgrade** — Conducts a lease database upgrade. This is useful when upgrading Kea.

- **lease-dump** — Dumps the contents of the lease database (for MySQL, PostgreSQL or CQL backends) to a CSV (comma separated values) text file. The first line of the file contains the column names. This is meant to be used as a diagnostic tool, so it provides a portable, human-readable form of the lease data.

backend specifies the backend type. Currently supported types are:

- **memfile** — Lease information is stored on disk in a text file.

- **mysql** — Lease information is stored in a MySQL relational database.

- **pgsql** — Lease information is stored in a PostgreSQL relational database.

- **cql** — Lease information is stored in a CQL database.

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

4.3 Supported Databases

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

Feature	Memfile
Status	Stable
Data format	CSV file
Leases	yes
Host Reservations	no
Options defined on per host basis	no

Table 4.1: List of available backends

4.3.1 memfile

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

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

4.3.1.1 Upgrading Memfile Lease Files from an Earlier Version of Kea

There are no special steps required to upgrade memfile lease files from an earlier version of Kea to a new version of Kea. During startup the servers will check the schema version of the lease files against their own. If there is a mismatch, the servers will automatically launch the LFC process to convert the files to the server's schema version. While this mechanism is primarily meant to ease the process of upgrading to newer versions of Kea, it can also be used for downgrading should the need arise. When upgrading, any values not present in the original lease files will be assigned appropriate default values. When downgrading, any data present in the files but not in the server's schema will be dropped. If you wish to convert the files manually, prior to starting the servers you may do so by running the LFC process yourself. See Chapter 11 for more information.

4.3.2 MySQL

MySQL is able to store leases, host reservations and options defined on a per host basis. This section can be safely ignored if you chose to store the data in other backends.

4.3.2.1 First Time Creation of the MySQL Database

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

To create the database:

1. Log into MySQL as "root":

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

2. Create the MySQL database:

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

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

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

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

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

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

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

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

5. Exit MySQL:

   ```
   mysql> quit
   Bye
   $
   ```

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

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

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

4.3.2.2 Upgrading a MySQL Database from an Earlier Version of Kea

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

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

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

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

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

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

4.3.3 PostgreSQL

A PostgreSQL database must be set up if you want Kea to store lease and other information in PostgreSQL. This step can be safely ignored if you are using other database backends.

4.3.3.1 First Time Creation of the PostgreSQL Database

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

4. Exit PostgreSQL:

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

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

   ```
   $ psql -d database-name -U user-name -f path-to-kea/share/kea/scripts/pgsql/ ←
       dhcpdb_create.pgsql
   Password for user user-name:
   CREATE TABLE
   CREATE INDEX
   CREATE INDEX
   CREATE TABLE
   ```

```
CREATE INDEX
CREATE TABLE
START TRANSACTION
INSERT 0 1
INSERT 0 1
INSERT 0 1
COMMIT
CREATE TABLE
START TRANSACTION
INSERT 0 1
COMMIT
$
```

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

If instead you encounter an error like:

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

... you will need to alter the PostgreSQL configuration. Kea uses password authentication when connecting to the database and must have the appropriate entries added to PostgreSQL's pg_hba.conf file. This file is normally located in the primary data directory for your PostgreSQL server. The precise path may vary 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
```

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

4.3.3.2 Initialize the PostgreSQL Database Using kea-admin

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

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

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

4.3.3.3 Upgrading a PostgreSQL Database from an Earlier Version of Kea

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

Use the following command to check the current schema version:

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

Use the following command to perform an upgrade:

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

4.3.4 CQL (Cassandra)

Cassandra, or Cassandra Query Language (CQL), is the newest backend added to Kea. Since it was added recently and has not undergone as much testing as other backends, it is considered experimental: please use with caution. The CQL backend is currently able to store leases only. The ability to store host reservations will likely be added some time in the future.

The CQL database must be properly set up if you want Kea to store information in CQL. This section can be safely ignored if you chose to store the data in other backends.

4.3.4.1 First Time Creation of the Cassandra Database

If you are setting up the CQL database for the first time, you need to create the keyspace area within CQL. This needs to be done manually: **kea-admin** is not able to do this for you.

To create the database:

1. Export CQLSH_HOST environemnt variable:

   ```
   $ export CQLSH_HOST=localhost
   ```

2. Log into CQL:

   ```
   $ cqlsh
   cql>
   ```

3. Create the CQL keyspace:

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

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

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

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

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

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

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

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

4.3.4.2 Upgrading a CQL Database from an Earlier Version of Kea

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

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

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

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

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

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

4.3.5 Using Read-Only Databases with Host Reservations

If a read-only database is used for storing host reservations, Kea must be explicitly configured to operate on the database in read-only mode. Sections Section 7.2.3.2 and Section 8.2.3.2 describe when such configuration may be reqired and how to configure Kea to operate using a read-only host database.

4.3.6 Limitations Related to the use of SQL Databases

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

Chapter 5

Kea Configuration

Kea is designed to allow different methods by which it can be configured, each method being implemented by a component known as a configuration backend. At present, only one such backend is available, that allowing configuration by means of a JSON file.

5.1 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 the -c parameter.

5.1.1 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 form of 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.

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-config": {
        "interfaces": [ "eth0" ],
        "dhcp-socket-type": "raw"
    },
    "valid-lifetime": 4000,
    "renew-timer": 1000,
    "rebind-timer": 2000,
    "subnet4": [{
        "pools": [ { "pool": "192.0.2.1-192.0.2.200" } ],
        "subnet": "192.0.2.0/24"
    }]
},
```

```
# DHCPv4 specific configuration ends here.

# DHCPv6 specific configuration starts here.
"Dhcp6": {
    "interfaces-config": {
        "interfaces": [ "eth1" ]
    },
    "preferred-lifetime": 3000,
    "valid-lifetime": 4000,
    "renew-timer": 1000,
    "rebind-timer": 2000,
    "subnet6": [{
        "pools": [ { "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 the 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 by an ellipsis.

5.1.2 Simplified Notation

It is sometimes convenient to refer to a specific element in the configuration hierarchy. Each hierarchy level is separated by a slash. If there is an array, a specific instance within that array is referenced by a number in square brackets (with numbering starting at zero). For example, in the above configuration the valid-lifetime in the Dhcp6 component can be referred to as Dhcp6/valid-lifetime and the pool in the first subnet defined in the DHCPv6 configuration as Dhcp6/subnet6[0]/pool.

Chapter 6

Managing Kea with keactrl

6.1 Overview

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

6.2 Command Line Options

keactrl is run as follows:

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

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

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

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

6.3 The keactrl Configuration File

Depending on requirements, not all of the available servers need 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 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. Modification of the logging severity in a configuration file, as described in Chapter 17, will have no effect as long as the `kea_verbose` is set to "yes". Setting it to "no" will cause the server to use the logging levels specified in the Kea configuration file for respective loggers. If no logging configuration is specified, the default settings will be used.

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

6.4 Commands

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

- **start** - starts selected servers.

- **stop** - stops all running servers.

- **reload** - triggers reconfiguration of the selected servers by sending the SIGHUP signal to them.

- **status** - returns the status of the servers (active or inactive) and the names of the configuration files in use.

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

Kea's servers create PID files upon startup. These files are used by keactrl to determine whether or not a given server is running. If one or more servers are running when the start command is issued, the output will look similar to the following:

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

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

The following command stops all servers:

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

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 are not running, an informational message is displayed as in the **stop** command output below.

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

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

```
$ keactrl reload
INFO/keactrl: Reloading kea-dhcp4...
INFO/keactrl: Reloading kea-dhcp6...
INFO/keactrl: Reloading kea-dhcp-ddns...
```

If any of the servers are not running, an informational message is displayed as in the **reload** command output below.

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

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
```

6.5 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 7

The DHCPv4 Server

7.1 Starting and Stopping the DHCPv4 Server

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

- **-c** *file* - specifies the configuration file. This is the only mandatory switch.

- **-d** - specifies whether the server logging should be switched to debug/verbose mode. In verbose mode, the logging severity and debuglevel specified in the configuration file are ignored and "debug" severity and the maximum debuglevel (99) are assumed. The flag is convenient, for temporarily switching the server into maximum verbosity, e.g. when debugging.

- **-p** *port* - specifies UDP port on which the server will listen. This is only useful during testing, as a DHCPv4 server listening on ports other than the standard ones will not be able to handle regular DHCPv4 queries.

- **-v** - prints out the Kea version and exits.

- **-V** - prints out the Kea extended version with additional parameters and exits. The listing includes the versions of the libraries dynamically linked to Kea.

- **-W** - prints out the Kea configuration report and exits. The report is a copy of the `config.report` file produced by **`./configure`**: it is embedded in the executable binary.

The `config.report` may also be accessed more directly. The following command may be used to extract this information. The binary **path** may be found in the install directory or in the `.libs` subdirectory in the source tree. For example `kea/src/bin/dhcp4/.libs/kea-dhcp4`.

```
strings path/kea-dhcp4 | sed -n 's/;;;; //p'
```

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.

During startup the server will attempt to create a PID file of the form: localstatedir/[conf name].kea-dhcp6.pid where:

- **localstatedir**: The value as passed into the build configure script. It defaults to "/usr/local/var". (Note that this value may be overridden at run time by setting the environment variable KEA_PIDFILE_DIR. This is intended primarily for testing purposes.)

- **conf name**: The configuration file name used to start the server, minus all preceding path and file extension. For example, given a pathname of "/usr/local/etc/kea/myconf.txt", the portion used would be "myconf".

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

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

7.2 DHCPv4 Server Configuration

7.2.1 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 is as follows:

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

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

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

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

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

}
```

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

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

The configuration starts in the first line with the initial opening curly bracket (or brace). Each configuration 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 the 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 well formed JSON. That means that the parameters for any given scope must be separated by a comma and there must not be a comma after the last parameter. When reordering a configuration file, keep in mind that moving a parameter to or from the last position in a given scope may also require moving the comma. The second caveat is that it is uncommon — although legal JSON — to repeat the same parameter multiple times. If that happens, the last occurrence of a given parameter in a given scope is used while all previous instances are ignored. This is unlikely to cause any confusion as there are no real life reasons to keep multiple copies of the same parameter in your configuration file.

Moving onto the DHCPv4 configuration elements, the first few elements define some global parameters. **valid-lifetime** defines for how long the addresses (leases) given out by the server are valid. If nothing changes, a client that got an address is allowed to use it for 4000 seconds. (Note that integer numbers are specified as is, without any quotes around them.) **renew-timer** and **rebind-timer** are values (also in seconds) that define T1 and T2 timers that govern when the client will begin the renewal and rebind procedures. Note 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 **interfaces-config** map specifies the server configuration concerning the network interfaces, on which the server should listen to the DHCP messages. The **interfaces** parameter specifies a list of network interfaces on which the server should listen. Lists are opened and closed with square brackets, with elements separated by commas. Had we wanted to listen on two interfaces, the **interfaces-config** would look like this:

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

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

Finally, we need to define a list of IPv4 subnets. This is the most important DHCPv4 configuration structure as the server uses that information to process clients' requests. It defines all subnets from which the server is expected to receive DHCP requests. The subnets are specified with the **subnet4** parameter. It is a list, so it starts and ends with square brackets. Each subnet definition in the list has several attributes associated with it, so it is a structure and is opened and closed with braces. At a minimum, a subnet definition has to have at least two parameters: **subnet** (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"
    },
    {
        "pools": [ { "pool": "192.0.4.1 - 192.0.4.254" } ],
        "subnet": "192.0.4.0/24"
    }
]
```

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

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

7.2.2 Lease Storage

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

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 store lease information in a CSV file rather than a database. As well as requiring less administration, an advantage of using a file for storage is that it eliminates a dependency on third-party database software.

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

- **persist**: controls whether the new leases and updates to existing leases are written to the file. It is strongly recommended that the value of this parameter is set to **true** at all times, during the server's normal operation. Not writing leases to disk 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. The value of **false** is mostly useful for performance testing purposes. The default value of the **persist** parameter is **true**, which enables writing lease updates to the lease file.

- **name**: specifies an absolute location of the lease file in which new leases and lease updates will be recorded. The default value for this parameter is **"[kea-install-dir]/var/kea/kea-leases4.csv"** .

- **lfc-interval**: specifies the interval in seconds, at which the server will perform a lease file cleanup (LFC). This removes redundant (historical) information from the lease file and effectively reduces the lease file size. The cleanup process is described in more detailed fashion further in this section. The default value of the **lfc-interval** is **0**, which disables the LFC.

An example configuration of the Memfile backend is presented below:

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

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

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

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

Lease file cleanup is performed by a separate process (in background) to avoid a performance impact on the server process. In order to avoid the conflicts between two processes both using the same lease files, the LFC process operates on the copy of the original lease file, rather than on the lease file used by the server to record lease updates. There are also other files being created as a side effect of the lease file cleanup. The detailed description of the LFC is located on the Kea wiki: http://kea.isc.org/wiki/LFCDesign.

7.2.2.2 Lease Database Configuration

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

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

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

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

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

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

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

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" : "", ... }, ... }
```

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

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

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

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.)

7.2.3   Hosts Storage

Kea is also able to store information about host reservations in the database. The hosts database configuration uses the same syntax as the lease database. In fact, a Kea server opens independent connections for each purpose, be it lease or hosts information. This arrangement gives the most flexibility. Kea can be used to keep leases and host reservations separately, but can also point to the same database. Currently the supported hosts database types are MySQL and PostgreSQL. The Cassandra backend does not support host reservations yet.

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

7.2.3.1   DHCPv4 Hosts Database Configuration

Hosts database configuration is controlled through the Dhcp4/hosts-database parameters. If enabled, the type of the database must be set to "mysql" or "postgresql". Other hosts backends may be added in later versions of Kea.

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

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

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

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

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

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

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

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

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

7.2.3.2   Using Read-Only Databases for Host Reservations

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

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

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

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

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

7.2.4 Interface Configuration

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Using UDP sockets automatically disables the reception of broadcast packets from directly connected clients. This effectively means that the UDP sockets can be used for relayed traffic only. When using the raw sockets, both the traffic from the directly connected clients and the relayed traffic will be handled. Caution should be taken when configuring the server to open multiple raw sockets on the interface with several IPv4 addresses assigned. If the directly connected client sends the message to the broadcast address all sockets on this link will receive this message and multiple responses will be sent to the client. Hence, the configuration with multiple IPv4 addresses assigned to the interface should not be used when the directly connected clients are operating on that link. To use a single address on such interface, the "interface-name/address" notation should be used.

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

7.2.5 Issues with Unicast Responses to DHCPINFORM

The use of UDP sockets has certain benefits in deployments where the server receives only relayed traffic; these benefits are mentioned in Section 7.2.4. From the administrator's perspective it is often desirable to configure the system's firewall to filter out the unwanted traffic, and the use of UDP sockets facilitates this. However, the administrator must also be aware of the implications related to filtering certain types of traffic as it may impair the DHCP server's operation.

In this section we are focusing on the case when the server receives the DHCPINFORM message from the client via a relay. According to RFC 2131, the server should unicast the DHCPACK response to the address carried in the "ciaddr" field. When the UDP socket is in use, the DHCP server relies on the low level functions of an operating system to build the data link, IP and UDP

layers of the outgoing message. Typically, the OS will first use ARP to obtain the client's link layer address to be inserted into the frame's header, if the address is not cached from a previous transaction that the client had with the server. When the ARP exchange is successful, the DHCP message can be unicast to the client, using the obtained address.

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

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

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

7.2.6 IPv4 Subnet Identifier

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

If there are multiple subnets configured with auto-generated identifiers and one of them is removed, the subnet identifiers may be renumbered. For example: if there are four subnets and the third is removed the last subnet will be assigned the identifier that the third subnet had before removal. As a result, the leases stored in the lease database for subnet 3 are now associated with subnet 4, something that may have unexpected consequences. It is planned to implement a 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 specified subnet identifier to the newly configured subnet:

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

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

7.2.7 Configuration of IPv4 Address Pools

The main role of a DHCPv4 server is address assignment. For this, the server has to be configured with at least one subnet and one pool of dynamic addresses for it to manage. 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.

Each **pool** is a structure that contains the parameters that describe a single pool. Currently there is only one parameter, **pool**, which gives the range of addresses in the pool. 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": [
        {
            "subnet": "192.0.2.0/24",
            "pools": [
                { "pool": "192.0.2.10-192.0.2.20" },
                { "pool": "192.0.2.64/26" }
            ],
            ...
        }
    ],
    ...
}
```

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

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

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

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

When configuring a DHCPv4 server using prefix/length notation, please pay attention to the boundary values. When specifying that the server can use a given pool, it will also be able to allocate the first (typically 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.

7.2.8 Standard DHCPv4 Options

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

Table 7.1 comprises the list of the standard DHCPv4 options whose values can be configured using the configuration structures described in this section. This table excludes the options which require special processing and thus cannot be configured with some fixed values. The last column of the table indicates which options can be sent by the server even when they are not requested in the Parameter Request list option, and those which are sent only when explicitly requested.

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

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

The **name** parameter specifies the option name. For a list of currently supported names, see Table 7.1 below. The **code** parameter specifies the option code, which must match one of the values from that list. The next line specifies the option space, which must always be set to "dhcp4" as these are standard DHCPv4 options. For other option spaces, including custom option spaces, see Section 7.2.11. The next line specifies the format in which the data will be entered: use of CSV (comma separated values) is recommended. The sixth line gives the actual value to be sent to clients. Data is specified as normal text, with values separated by commas if more than one value is allowed.

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

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

Most of the parameters in the "option-data" structure are optional and can be omitted in some circumstances as discussed in the Section 7.2.12.

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

The currently supported standard DHCPv4 options are listed in Table 7.1 and Table 7.2. 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 7.3.

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 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 currently does not provide a definition. In order to use such options, a server administrator must create a definition as described in Section 7.2.9 in the 'dhcp4' option space. This definition should match the option format described in the relevant RFC but the configuration mechanism will allow any option format as it presently has no means to validate it.

7.2.9 Custom DHCPv4 options

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

```
"Dhcp4": {
    "option-def": [
        {
            "name": "foo",
            "code": 222,
            "type": "uint32",
            "array": false,
            "record-types": "",
            "space": "dhcp4",
            "encapsulate": ""
        }, ...
    ],
    ...
}
```

The **false** value of the **array** parameter determines that the option does NOT comprise an array of "uint32" values but is, instead, a single value. Two other parameters have been left blank: **record-types** and **encapsulate**. The former specifies the comma separated list of option data fields if the option comprises a record of data fields. This should be non-empty if the **type** is set to

Name	Code	Type	Array?	Returned if not requested?
time-offset	2	int32	false	false
routers	3	ipv4-address	true	true
time-servers	4	ipv4-address	true	false
name-servers	5	ipv4-address	true	false
domain-name-servers	6	ipv4-address	true	true
log-servers	7	ipv4-address	true	false
cookie-servers	8	ipv4-address	true	false
lpr-servers	9	ipv4-address	true	false
impress-servers	10	ipv4-address	true	false
resource-location-servers	11	ipv4-address	true	false
boot-size	13	uint16	false	false
merit-dump	14	string	false	false
domain-name	15	fqdn	false	true
swap-server	16	ipv4-address	false	false
root-path	17	string	false	false
extensions-path	18	string	false	false
ip-forwarding	19	boolean	false	false
non-local-source-routing	20	boolean	false	false
policy-filter	21	ipv4-address	true	false
max-dgram-reassembly	22	uint16	false	false
default-ip-ttl	23	uint8	false	false
path-mtu-aging-timeout	24	uint32	false	false
path-mtu-plateau-table	25	uint16	true	false
interface-mtu	26	uint16	false	false
all-subnets-local	27	boolean	false	false
broadcast-address	28	ipv4-address	false	false
perform-mask-discovery	29	boolean	false	false
mask-supplier	30	boolean	false	false
router-discovery	31	boolean	false	false
router-solicitation-address	32	ipv4-address	false	false
static-routes	33	ipv4-address	true	false
trailer-encapsulation	34	boolean	false	false
arp-cache-timeout	35	uint32	false	false
ieee802-3-encapsulation	36	boolean	false	false
default-tcp-ttl	37	uint8	false	false
tcp-keepalive-interval	38	uint32	false	false
tcp-keepalive-garbage	39	boolean	false	false

Table 7.1: List of standard DHCPv4 options

Name	Code	Type	Array?	Returned if not requested?
nis-domain	40	string	false	false
nis-servers	41	ipv4-address	true	false
ntp-servers	42	ipv4-address	true	false
vendor-encapsulated-options	43	empty	false	false
netbios-name-servers	44	ipv4-address	true	false
netbios-dd-server	45	ipv4-address	true	false
netbios-node-type	46	uint8	false	false
netbios-scope	47	string	false	false
font-servers	48	ipv4-address	true	false
x-display-manager	49	ipv4-address	true	false
dhcp-option-overload	52	uint8	false	false
dhcp-message	56	string	false	false
dhcp-max-message-size	57	uint16	false	false
vendor-class-identifier	60	binary	false	false
nwip-domain-name	62	string	false	false
nwip-suboptions	63	binary	false	false
nisplus-domain-name	64	string	false	false
nisplus-servers	65	ipv4-address	true	false
tftp-server-name	66	string	false	false
boot-file-name	67	string	false	false
mobile-ip-home-agent	68	ipv4-address	true	false
smtp-server	69	ipv4-address	true	false
pop-server	70	ipv4-address	true	false
nntp-server	71	ipv4-address	true	false
www-server	72	ipv4-address	true	false
finger-server	73	ipv4-address	true	false
irc-server	74	ipv4-address	true	false
streettalk-server	75	ipv4-address	true	false
streettalk-directory-assistance-server	76	ipv4-address	true	false
user-class	77	binary	false	false
client-system	93	uint16	true	false
client-ndi	94	record (uint8, uint8, uint8)	false	false
uuid-guid	97	record (uint8, binary)	false	false
subnet-selection	118	ipv4-address	false	false
domain-search	119	binary	false	false
vivco-suboptions	124	binary	false	false
vivso-suboptions	125	binary	false	false

Table 7.2: 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 7.3: List of standard DHCP option types

"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.

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

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

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

New options can take more complex forms than simple use of primitives (uint8, string, ipv4-address etc): it is possible to define an option comprising a number of existing primitives. Assume we want to define a new option that will consist of an IPv4 address, followed by an unsigned 16 bit integer, followed by a boolean value, followed by a text string. Such an option could be defined in the following way:

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

```
        ...
}
```

The **type** is set to "record" to indicate that the option contains multiple values of different types. These types are given as a comma-separated list in the **record-types** field and should be ones from those listed in Table 7.3.

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 to **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 the 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 versions of Kea will accept all those values for all boolean parameters.

7.2.10 DHCPv4 Vendor Specific Options

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

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

```
"Dhcp4": {
    "option-def": [
        {
            "name": "foo",
            "code": 1,
            "space": "vendor-encapsulated-options-space",
            "type": "record",
            "array": false,
            "record-types": "ipv4-address, uint16, string",
            "encapsulate": ""
        }
    ],
    ...
}
```

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

```
"Dhcp4": {
    "option-data": [
        {
```

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

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

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

Alternatively, the option can be specified using its code.

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

7.2.11 Nested DHCPv4 Options (Custom Option Spaces)

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

Note that 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 7.2.10).

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

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

```
                "record-types": "",
                "array": false,
                "encapsulate": ""
            }
        ],
        ...
}
```

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

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

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

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

Finally, we can set values for the new options:

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

Note that it is possible to create an option which carries some data in addition to the sub-options defined in the encapsulated option space. For example, if the "container" option from the previous example 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 Unspecified Parameters for DHCPv4 Option Configuration

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

- **name** - the server requires an option name or option code to identify an option. If this parameter is unspecified, the option code must be specified.

- **code** - the server requires an option name or option code to identify an option. This parameter may be left unspecified if the **name** parameter is specified. However, this also requires that the particular option has its definition (it is either a standard option or an administrator created a definition for the option using an 'option-def' structure), as the option definition associates an option with a particular name. It is possible to configure an option for which there is no definition (unspecified option format). Configuration of such options requires the use of option code.

- **space** - if the option space is unspecified it will default to 'dhcp4' which is an option space holding DHCPv4 standard options.

- **data** - if the option data is unspecified it defaults to an empty value. The empty value is mostly used for the options which have no payload (boolean options), but it is legal to specify empty values for some options which carry variable length data and which the specification allows for the length of 0. For such options, the data parameter may be omitted in the configuration.

- **csv-format** - if this value is not specified and the definition for the particular option exists, the server will assume that the option data is specified as a list of comma separated values to be assigned to individual fields of the DHCP option. If the definition does not exist for this option, the server will assume that the data parameter contains the option payload in the binary format (represented as a string of hexadecimal digits). Note that not specifying this parameter doesn't imply that it defaults to a fixed value, but the configuration data interpretation also depends on the presence of the option definition. An administrator must be aware if the definition for the particular option exists when this parameter is not specified. It is generally recommended to not specify this parameter only for the options for which the definition exists, e.g. standard options. Setting **csv-format** to an explicit value will cause the server to strictly check the format of the option data specified.

7.2.13 Stateless Configuration of DHCPv4 Clients

The DHCPv4 server supports the stateless client configuration whereby the client has an IP address configured (e.g. using manual configuration) and only contacts the server to obtain other configuration parameters, e.g. addresses of DNS servers. In order to obtain the stateless configuration parameters the client sends the DHCPINFORM message to the server with the "ciaddr" set to the address that the client is currently using. The server unicasts the DHCPACK message to the client that includes the stateless configuration ("yiaddr" not set).

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

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

This subnet specifies the single option which will be included in the DHCPACK message to the client in response to DHCPIN-FORM. 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.

7.2.14 Client Classification in DHCPv4

The DHCPv4 server includes support for client classification. For a deeper discussion of the classification process see Chapter 12.

In certain cases it is useful to differentiate between different types of clients and treat them accordingly. It is envisaged that client classification will be used for changing the behavior of almost any part of the DHCP message processing, including the assignment of leases from different pools, the assignment of different options (or different values of the same options) etc. In the current release of the software however, there are only three mechanisms that take advantage of client classification: subnet selection, assignment of different options, and, for cable modems, there are specific options for use with the TFTP server address and the boot file field.

Kea can be instructed to limit access to given subnets based on class information. This is particularly useful for cases where two types of devices share the same link and are expected to be served from two different subnets. The primary use case for such a scenario is cable networks. Here, there are two classes of devices: the cable modem itself, which should be handed a lease from subnet A and all other devices behind the modem 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 12.6.

The process of doing classification is conducted in three steps. The first step is to assess an incoming packet and assign it to zero or more classes. The second step is to choose a subnet, possibly based on the class information. The third step is to assign options, again possibly based on the class information.

There are two methods of doing classification. The first is automatic and relies on examining the values in the vendor class options. Information from these options is extracted and a class name is constructed from it and added to the class list for the packet. The second allows you to specify an expression that is evaluated for each packet. If the result is true the packet is a member of the class.

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

7.2.14.1 Setting Fixed Fields in Classification

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

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

```
"Dhcp4": {
    "client-classes": [
        {
            "name": "ipxe_efi_x64",
            "test": "option[93].hex == 0x0009",
```

```
              "next-server": "192.0.2.254",
              "server-hostname": "hal9000",
              "boot-file-name": "/dev/null"
          },
          ...
      ],
      ...
            }
```

If there are multiple classes defined and an incoming packet is matched to multiple classes, the class whose name is alphabetically the first is used.

7.2.14.2 Using Vendor Class Information in Classification

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

Note
Kea 1.0 and earlier versions performed special actions for clients that were in VENDOR_CLASS_docsis3.0. This is no longer the case in Kea 1.1 and later. In these versions the old behavior can be achieved by defining VENDOR_CLASS_docsis3.0 and setting its next-server and boot-file-name values appropriately.

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

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

7.2.14.3 Defining and Using Custom Classes

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

```
"Dhcp4": {
    "client-classes": [
        {
            "name": "Client_foo",
            "test": "substring(option[61].hex,0,3) == 'foo'",
            "option-data": [
                {
                    "name": "domain-name-servers",
                    "code": 6,
                    "space": "dhcp4",
                    "csv-format": true,
```

```
                            "data": "192.0.2.1, 192.0.2.2"
                    }
                ]
        },
        ...
    ],
    "subnet4": [
        {
            "subnet": "192.0.2.0/24",
            "pools": [ { "pool": "192.0.2.10 - 192.0.2.20" } ],
            "client-class": "Client_foo"
        },
        ...
    ],
    ...
}
```

7.2.15 DDNS for DHCPv4

As mentioned earlier, kea-dhcp4 can be configured to generate requests to the DHCP-DDNS server (referred to here as "D2") to update DNS entries. These requests are known as 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 D2 are contained in the **dhcp-ddns** section of the kea-dhcp4 server configuration. The mandatory parameters for the DHCP DDNS configuration are **enable-updates** which is unconditionally required, and **qualifying-suffix** which has no default value and is required when **enable-updates** is set to **true**. The two (disabled and enabled) minimal DHCP DDNS configurations are:

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

and for example:

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

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

- **"server-ip": "127.0.0.1"**

- **"server-port": 53001**

- **"sender-ip": ""**

- **"sender-port": 0**

- **"max-queue-size": 1024**

- **"ncr-protocol": "UDP"**

- **"ncr-format": "JSON"**

- **"override-no-update": false**

- **"override-client-update": false**

- **"replace-client-name": "never"**

- **"generated-prefix": "myhost"**

7.2.15.1 DHCP-DDNS Server Connectivity

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

- **enable-updates** - determines whether 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 D2 listens for requests. The default is the local loopback interface at address 127.0.0.1. You may specify either an IPv4 or IPv6 address.

- **server-port** - port on which D2 listens for requests. The default value is 53001.

- **sender-ip** - IP address which kea-dhcp4 should use to send requests to D2. The default value is blank which instructs kea-dhcp4 to select a suitable address.

- **sender-port** - port which kea-dhcp4 should use to send requests to D2. The default value of 0 instructs kea-dhcp4 to select a suitable port.

- **max-queue-size** - maximum number of requests allowed to queue waiting to be sent to D2. This value guards against requests accumulating uncontrollably if they are being generated faster than they can be delivered. If the number of requests queued for transmission reaches this value, DDNS updating will be turned off until the queue backlog has been sufficiently reduced. The intention is to allow the kea-dhcp4 server to continue lease operations without running the risk that its memory usage grows without limit. The default value is 1024.

- **ncr-protocol** - socket protocol use when sending requests to D2. Currently only UDP is supported. TCP may be available in an upcoming release.

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

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

7.2.15.2 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 D2 in the form of NCRs. Carrying out the actual DNS updates and dealing with such things as conflict resolution are within the purview of D2 itself (Chapter 10). 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. 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 7.4: 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 D2 server to update only reverse DNS data. The parameter **override-client-update** can be used to instruct the server to override client delegation requests. When this parameter is true, kea-dhcp4 will disregard requests for client delegation and generate a DDNS request to update both forward and reverse DNS data. In this case, the N-S-O flags in the server's response to the client will be 0-1-1 respectively.

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

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

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

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

To override client delegation, 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.

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

1. If the DHCPREQUEST contains the client FQDN option, the candidate name is taken from there, otherwise it is taken from the Host Name option.

2. If the candidate name is a partial (i.e. unqualified) name then add a configurable suffix to the name and use the result as the FQDN.

3. If the candidate name provided is empty, generate a FQDN using a configurable prefix and suffix.

4. If the client provided neither option, then no DNS action will be taken.

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

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

- **always** - Replace the name the client sent. If the client sent no name, generate one for the client.

- **when-present** - Replace the name the client sent. If the client sent no name, do not generate one.

- **when-not-present** - Use the name the client sent. If the client sent no name, generate one for the client.

Note Note that formerly, this parameter was a boolean and permitted only values of **true** and **false**. Boolean values will still be accepted but may eventually be deprecated. A value of **true** equates to **when-present**, **false** equates to **never**.

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

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

The prefix used in the generation of 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. This parameter has no default value, thus it is mandatory when DDNS updates are enabled. To set its value simply set it to the desired string:

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

When generating a name, kea-dhcp4 will construct 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 the lease address is 172.16.1.10, the qualifying suffix "example.com", and the default value is used for **generated-prefix**, the generated FQDN would be:

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

7.2.16 Next Server (siaddr)

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

7.2.17 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 states that the server must send client-id if the client sent it. That is Kea's default behavior. However, in some cases older devices that do not support RFC 6842. may refuse to accept responses that include the client-id option. To enable backward compatibility, an optional configuration parameter has been introduced. To configure it, use the following configuration statement:

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

7.2.18 Using Client Identifier and Hardware Address

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

- When the client contacts the server to allocate a new lease, the server must store the client identification information in the lease database as a search key.

- When the client is trying to renew or release the existing lease, the server must be able to find the existing lease entry in the database for this client, using the client identification information as a search key.

- Some configurations use static reservations for the IP addresses and other configuration information. The server's administrator uses client identification information to create these static assignments.

- In the dual stack networks there is often a need to correlate the lease information stored in DHCPv4 and DHCPv6 server for a particular host. Using common identification information by the DHCPv4 and DHCPv6 client allows the network administrator to achieve this correlation and better administer the network.

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

RFC 2131 indicates that the server may use both of these identifiers to identify the client but the "client identifier", if present, takes precedence over "chaddr". One of the reasons for this is that "client identifier" is independent from the hardware used by the client to communicate with the server. For example, if the client obtained the lease using one network card and then the network card is moved to another host, the server will wrongly identify this host is the one which has obtained the lease. Moreover, RFC 4361 gives the recommendation to use a DUID (see RFC 3315, the DHCPv6 specification) carried as "client identifier" when dual stack networks are in use to provide consistent identification information of the client, regardless of the protocol type it is using. Kea adheres to these specifications and the "client identifier" by default takes precedence over the value carried in "chaddr" field when the server searches, creates, updates or removes the client's lease.

When the server receives a DHCPDISCOVER or DHCPREQUEST message from the client, it will try to find out if the client already has a lease in the database and will hand out that lease rather than allocate a new one. Each lease in the lease database is associated with the "client identifier" and/or "chaddr". The server will first use the "client identifier" (if present) to search the lease. If the lease is found, the server will treat this lease as belonging to the client even if the current "chaddr" and the "chaddr" associated with the lease do not match. This facilitates the scenario when the network card on the client system has been replaced and thus the new MAC address appears in the messages sent by the DHCP client. If the server fails to find the lease using the "client identifier" it will perform another lookup using the "chaddr". If this lookup returns no result, the client is considered as not having a lease and the new lease will be created.

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

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

```
"Dhcp4": {
    ...
```

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

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

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

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

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

7.2.19 DHCPv4-over-DHCPv6: DHCPv4 Side

The support of DHCPv4-over-DHCPv6 transport is described in RFC 7341 and is implemented using cooperating DHCPv4 and DHCPv6 servers. This section is about the configuration of the DHCPv4 side (the DHCPv6 side is described in Section 8.2.20).

Note DHCPv4-over-DHCPv6 support is experimental and the details of the inter-process communication can change: both the DHCPv4 and DHCPv6 sides should be running the same version of Kea.

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

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

- **4o6-subnet**: Takes a prefix (i.e., an IPv6 address followed by a slash and a prefix length) which is matched against the source address.

- **4o6-interface-id**: Takes a relay interface ID option value.

- **4o6-interface**: Takes an interface name which is matched against the incoming interface name.

The following configuration was used during some tests:

```
{

# DHCPv4 conf
"Dhcp4": {
    "interfaces-config": {
        "interfaces": [ "eno33554984" ]
    },

    "lease-database": {
        "type": "memfile",
        "name": "leases4"
    },

    "valid-lifetime": 4000,

    "subnet4": [ {
        "subnet": "10.10.10.0/24",
        "4o6-interface": "eno33554984",
        "4o6-subnet": "2001:db8:1:1::/64",
        "pools": [ { "pool": "10.10.10.100 - 10.10.10.199" } ]
    } ],

    "dhcp4o6-port": 6767

},

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

}
```

7.3 Host Reservation in DHCPv4

There are many cases where it is useful to provide a configuration on a per host basis. The most obvious one is to reserve a specific, static address for exclusive use by a given client (host) - the returning client will receive the same address from the server every time, and other clients will generally not receive that address. Another example when the host reservations are applicable is when a host has specific requirements, e.g. a printer that needs additional DHCP options. Yet another possible use case is to define unique names for hosts.

Note that there may be cases when the new reservation has been made for the client for the address being currently in use by another client. We call this situation a "conflict". The conflicts get resolved automatically over time as described in subsequent sections. Once the conflict is resolved, the client will keep receiving the reserved configuration when it renews.

Host reservations are defined as parameters for each subnet. Each host has to be identified by an identifier, for example the hardware/MAC address. There is an optional **reservations** array in the **Subnet4** element. Each element in that array is a structure that holds information about reservations for a single host. In particular, the structure has to have an identifier that uniquely identifies a host. In the DHCPv4 context, the identifier is usually a hardware or MAC address. In most cases an IP address will be specified. It is also possible to specify a hostname, host specific options or fields carried within DHCPv4 message such as siaddr, sname or file.

In Kea 1.0.0 it was only possible to create host reservations using client's hardware address. Host reservations by client identifier, DUID and circuit-id have been added in Kea 1.1.0.

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

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

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

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

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

Adding host reservation incurs a performance penalty. In principle, when a server that does not support host reservation responds to a query, it needs to check whether there is a lease for a given address being considered for allocation or renewal. The server that also supports host reservation has to perform additional checks: not only if the address is currently used (i.e. if there is a lease for it), but also whether the address could be used by someone else (i.e. there is a reservation for it). That additional check incurs additional overhead.

7.3.1 Address Reservation Types

In a typical scenario there is an IPv4 subnet defined, e.g. 192.0.2.0/24, with certain part of it dedicated for dynamic allocation by the DHCPv4 server. That dynamic part is referred to as a dynamic pool or simply a pool. In principle, a host reservation can

reserve any address that belongs to the subnet. The reservations that specify addresses that belong to configured pools are called "in-pool reservations". In contrast, those that do not belong to dynamic pools are called "out-of-pool reservations". There is no formal difference in the reservation syntax and both reservation types are handled uniformly. However, upcoming releases may offer improved performance if there are only out-of-pool reservations as the server will be able to skip reservation checks when dealing with existing leases. Therefore, system administrators are encouraged to use out-of-pool reservations if possible.

7.3.2 Conflicts in DHCPv4 Reservations

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

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

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

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

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

7.3.3 Reserving a Hostname

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

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

```
{
    "subnet4": [ {
        "subnet": "10.0.0.0/24",
        "pools": [ { "pool": "10.0.0.10-10.0.0.100" } ],
        "reservations": [
```

```
            {
               "hw-address": "aa:bb:cc:dd:ee:ff",
               "hostname": "alice-laptop"
            }
        ]
    }],
    "dhcp-ddns": {
        "enable-updates": true,
        "qualifying-suffix": "example.isc.org."
    }
}
```

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

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

        ]
    }],
    "dhcp-ddns": {
        "enable-updates": true,
    }
}
```

7.3.4 Including Specific DHCPv4 Options in Reservations

Kea 1.1.0 introduced the ability to specify options on a per host basis. The options follow the same rules as any other options. These can be standard options (see Section 7.2.8), custom options (see Section 7.2.9) or vendor specific options (see Section 7.2.10). The following example demonstrates how standard options can be defined.

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

```
    } ]
}
```

Vendor specific options can be reserved in a similar manner:

```
{
    "subnet4": [ {
        "reservations": [
        {
            "hw-address": "aa:bb:cc:dd:ee:ff",
            "ip-address": "10.0.0.7",
            "option-data": [
            {
                "name": "vivso-suboptions",
                "data": "4491"
            },
            {

                "name": "tftp-servers",
                "space": "vendor-4491",
                "data": "10.1.1.202,10.1.1.203"
            } ]
        } ]
    } ]
}
```

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

7.3.5 Reserving Next Server, Server Hostname and Boot File Name

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

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

```
{
    "subnet4": [ {
        "reservations": [
        {
            "hw-address": "aa:bb:cc:dd:ee:ff",
            "next-server": "10.1.1.2",
            "server-hostname": "server-hostname.example.org",
            "boot-file-name": "/tmp/bootfile.efi"
        } ]
    } ]
}
```

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

7.3.6 Reserving Client Classes in DHCPv4

Section 12.4 explains how to configure the server to assign classes to a client based on the content of the options that this client sends to the server. Host reservations mechanisms also allow for statically assigning classes to the clients. The definitions of these classes must exist in the Kea configuration. The following configuration snippet shows how to specify that a client belongs to classes **reserved-class1** and **reserved-class2**. Those classes are associated with specific options being sent to the clients which belong to them.

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

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

        }
        ]
    } ]
}
```

Static class assignments, as shown above, can be used in conjuction with classification using expressions.

7.3.7 Storing Host Reservations in MySQL or PostgreSQL

It is possible to store host reservations in MySQL or PostgreSQL database. See Section 7.2.3 for information on how to configure Kea to use reservations stored in MySQL or PostgreSQL. Kea does not provide any dedicated tools for managing reservations in a database. The Kea wiki http://kea.isc.org/wiki/HostReservationsHowTo provides detailed information and examples of how reservations can be inserted into the database.

Note
In Kea 1.1.0 maximum length of an option specified per host is arbitrarily set to 4096 bytes.

7.3.8 Storing host reservations in CQL (Cassandra)

Kea currently does not support storing reservations in Cassandra (CQL).

7.3.9 Fine Tuning DHCPv4 Host Reservation

The host reservation capability introduces additional restrictions for the allocation engine (the component of Kea that selects an address for a client) during lease selection and renewal. In particular, three major checks are necessary. First, when selecting a

new lease, it is not sufficient for a candidate lease to not be used by another DHCP client. It also must not be reserved for another client. Second, when renewing a lease, additional check must be performed whether the address being renewed is not reserved for another client. Finally, when a host renews an address, the server has to check whether there is a reservation for this host, so the existing (dynamically allocated) address should be revoked and the reserved one be used instead.

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

- **all** - enables all host reservation types. This is the default value. This setting is the safest and the most flexible. It allows in-pool and out-of-pool reservations. As all checks are conducted, it is also the slowest.

- **out-of-pool** - allows only out of pool host reservations. With this setting in place, the server may assume that all host reservations are for addresses that do not belong to the dynamic pool. Therefore it can skip the reservation checks when dealing with in-pool addresses, thus improving performance. Do not use this mode if any of your reservations use in-pool address. Caution is advised when using this setting: Kea 1.1.0 does not sanity check the reservations against **reservation-mode** and misconfiguration may cause problems.

- **disabled** - host reservation support is disabled. As there are no reservations, the server will skip all checks. Any reservations defined will be completely ignored. As the checks are skipped, the server may operate faster in this mode.

An example configuration that disables reservation looks like follows:

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

Another aspect of the host reservations are the different types of identifiers. Kea 1.1.0 supports four types of identifiers (hw-address, duid, client-id and circuit-id), but more identifier types are likely to be added in the future. This is beneficial from a usability perspective. However, there is a drawback. For each incoming packet Kea has to to extract each identifier type and then query the database to see if there is a reservation done by this particular identifier. If nothing is found, the next identifier is extracted and the next query is issued. This process continues until either a reservation is found or all identifier types have been checked. Over time with an increasing number of supported identifier types, Kea would become slower and slower.

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

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

If not specified, the default value is:

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

7.4 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.

7.5 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 directly connected clients, the server will check the configuration for the interface on which the message has been received and, if the server configuration doesn't match any configured subnet, the message is discarded.

Assuming that the server's interface is configured with the IPv4 address 192.0.2.3, the server will only process messages received through this interface from a directly connected client if there is a subnet configured to which this IPv4 address belongs, e.g. 192.0.2.0/24. The server will use this subnet to assign IPv4 address for the client.

The rule above does not apply when the client unicasts its message, i.e. is trying to renew its lease. Such a message is accepted through any interface. The renewing client sets ciaddr to the currently used IPv4 address. 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 7.5.1 for details.

Note

The subnet selection mechanism described in this section is based on the assumption that client classification is not used. The classification mechanism alters the way in which a subnet is selected for the client, depending on the classes to which the client belongs.

7.5.1 Using a Specific Relay Agent for a Subnet

A 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 from which the server will assign addresses. In the typical case, the server is able to use the IPv4 address inserted by the relay (in the giaddr field of the DHCPv4 packet) to select the appropriate subnet.

However, that is not always the case. In certain uncommon — but valid — deployments, the relay address may not match the subnet. This usually means that there is more than one subnet allocated for a given link. The two most common examples where this is the case are long lasting network renumbering (where both old and new address space is still being used) and a cable network. In a cable network both cable modems and the devices behind them are physically connected to the same link, yet they use distinct addressing. In such a case, the DHCPv4 server needs additional information (the IPv4 address of the relay) to properly select an appropriate subnet.

The following example assumes that there is a subnet 192.0.2.0/24 that is accessible via a relay that uses 10.0.0.1 as its IPv4 address. The server 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"
            },
            ...
        }
    ],
    ...
}
```

7.5.2 Segregating IPv4 Clients in a Cable Network

In certain cases, it is useful to mix relay address information, introduced in Section 7.5.1 with client classification, explained in Chapter 12. One specific example is cable network, where typically modems get addresses from a different subnet than all devices connected behind them.

Let us assume that there is one CMTS (Cable Modem Termination System) with one CM MAC (a physical link that modems are connected to). We want the modems to get addresses from the 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"
            }
        }
    ],
    ...
}
```

7.6 Duplicate Addresses (DHCPDECLINE Support)

The DHCPv4 server is configured with a certain pool of addresses that it is expected to hand out to the DHCPv4 clients. It is assumed that the server is authoritative and has complete jurisdiction over those addresses. However, due to various reasons, such as misconfiguration or a faulty client implementation that retains its address beyond the valid lifetime, there may be devices connected that use those addresses without the server's approval or knowledge.

Such an unwelcome event can be detected by legitimate clients (using ARP or ICMP Echo Request mechanisms) and reported to the DHCPv4 server using a DHCPDECLINE message. The server will do a sanity check (if the client declining an address really was supposed to use it), and then will conduct a clean up operation. Any DNS entries related to that address will be removed, the fact will be logged and hooks will be triggered. After that is done, the address will be marked as declined (which indicates that it is used by an unknown entity and thus not available for assignment to anyone) and a probation time will be set on it. Unless otherwise configured, the probation period lasts 24 hours. After that period, the server will recover the lease (i.e. put it back into the available state) and the address will be available for assignment again. It should be noted that if the underlying issue of a

misconfigured device is not resolved, the duplicate address scenario will repeat. On the other hand, it provides an opportunity to recover from such an event automatically, without any sysadmin intervention.

To configure the decline probation period to a value other than the default, the following syntax can be used:

```
  "Dhcp4": {
    "decline-probation-period": 3600,
    "subnet4": [ ... ],
    ...
}
```

The parameter is expressed in seconds, so the example above will instruct the server to recycle declined leases after an hour.

There are several statistics and hook points associated with the Decline handling procedure. The lease4_decline hook is triggered after the incoming DHCPDECLINE message has been sanitized and the server is about to decline the lease. The declined-addresses statistic is increased after the hook returns (both global and subnet specific variants). (See Section 7.7 and Chapter 13 for more details on DHCPv4 statistics and Kea hook points.)

Once the probation time elapses, the declined lease is recovered using the standard expired lease reclamation procedure, with several additional steps. In particular, both declined-addresses statistics (global and subnet specific) are decreased. At the same time, reclaimed-declined-addresses statistics (again in two variants, global and subnet specific) are increased.

Note about statistics: The server does not decrease the assigned-addresses statistics when a DHCPDECLINE is received and processed successfully. While technically a declined address is no longer assigned, the primary usage of the assigned-addresses statistic is to monitor pool utilization. Most people would forget to include declined-addresses in the calculation, and simply do assigned-addresses/total-addresses. This would have a bias towards under-representing pool utilization. As this has a potential for major issues, we decided not to decrease assigned addresses immediately after receiving DHCPDECLINE, but to do it later when we recover the address back to the available pool.

7.7 Statistics in the DHCPv4 Server

> **Note**
> This section describes DHCPv4-specific statistics. For a general overview and usage of statistics, see Chapter 14.

The DHCPv4 server supports the following statistics:

7.8 Management API for the DHCPv4 Server

The management API allows the issuing of specific management commands, such as statistics retrieval, reconfiguration or shutdown. For more details, see Chapter 15. Currently the only supported communication channel type is UNIX stream socket. By default there are no sockets open. To instruct Kea to open a socket, the following entry in the configuration file can be used:

```
"Dhcp4": {
    "control-socket": {
        "socket-type": "unix",
        "socket-name": "/path/to/the/unix/socket"
    },

    "subnet4": [
        ...
    ],
    ...
}
```

Statistic	Data Type	Description
pkt4-received	integer	Number of DHCPv4 packets received. This includes all packets: valid, bogus, corrupted, rejected etc. This statistic is expected to grow rapidly.
pkt4-discover-received	integer	Number of DHCPDISCOVER packets received. This statistic is expected to grow. Its increase means that clients that just booted started their configuration process and their initial packets reached your server.
pkt4-offer-received	integer	Number of DHCPOFFER packets received. This statistic is expected to remain zero at all times, as DHCPOFFER packets are sent by the server and the server is never expected to receive them. Non-zero value indicates an error. One likely cause would be a misbehaving relay agent that incorrectly forwards DHCPOFFER messages towards the server, rather back to the clients.
pkt4-request-received	integer	Number of DHCPREQUEST packets received. This statistic is expected to grow. Its increase means that clients that just booted received server's response (DHCPOFFER), accepted it and now requesting an address (DHCPREQUEST).
pkt4-ack-received	integer	Number of DHCPACK packets received. This statistic is expected to remain zero at all times, as DHCPACK packets are sent by the server and the server is never expected to receive them. Non-zero value indicates an error. One likely cause would be a misbehaving relay agent that incorrectly forwards DHCPACK messages towards the server, rather back to the clients.
pkt4-nak-received	integer	Number of DHCPNAK packets received. This statistic is expected to remain zero at all times, as DHCPNAK packets are sent by the server and the server is never expected to receive them. Non-zero value indicates an error. One likely cause would be a misbehaving relay agent that incorrectly forwards DHCPNAK messages towards the server, rather back to the clients.
pkt4-release-received	integer	Number of DHCPRELEASE packets received. This statistic is expected to grow. Its increase means that clients that had an address are shutting down or stop using their addresses.
pkt4-decline-received	integer	Number of DHCPDECLINE packets received. This statistic is expected to remain close to zero. Its increase means that a client that leased an address, but discovered that the address is currently used by an unknown device in your network.
		Number of DHCPINFORM packets

The length of the path specified by the **socket-name** parameter is restricted by the maximum length for the unix socket name on your operating system, i.e. the size of the **sun_path** field in the **sockaddr_un** structure, decreased by 1. This value varies on different operating systems between 91 and 107 characters. Typical values are 107 on Linux and 103 on FreeBSD.

Communication over control channel is conducted using JSON structures. See the Control Channel section in the Kea Developer's Guide for more details.

The DHCPv4 server supports **statistic-get**, **statistic-reset**, **statistic-remove**, **statistic-get-all**, **statistic-reset-all** and **statistic-remove-all**, specified in Section 14.3. It also supports **list-commands** and **shutdown**, specified in Section 15.3.2 and Section 15.3.3, respectively.

7.9 Supported DHCP Standards

The following standards are currently supported:

- *Dynamic Host Configuration Protocol*, RFC 2131: Supported messages are DHCPDISCOVER (1), DHCPOFFER (2), DHCPRE-QUEST (3), DHCPRELEASE (7), DHCPINFORM (8), DHCPACK (5), and DHCPNAK(6).

- *DHCP Options and BOOTP Vendor Extensions*, RFC 2132: Supported options are: PAD (0), END(255), Message Type(53), DHCP Server Identifier (54), Domain Name (15), DNS Servers (6), IP Address Lease Time (51), Subnet mask (1), and Routers (3).

- *DHCP Relay Agent Information Option*, RFC 3046: Relay Agent Information option is supported.

- *Vendor-Identifying Vendor Options for Dynamic Host Configuration Protocol version 4*, RFC 3925: Vendor-Identifying Vendor Class and Vendor-Identifying Vendor-Specific Information options are supported.

- *Client Identifier Option in DHCP Server Replies*, RFC 6842: Server by default sends back client-id option. That capability may be disabled. See Section 7.2.17 for details.

7.10 DHCPv4 Server Limitations

These are the current limitations of the DHCPv4 server software. Most of them are reflections of the current stage of development and should be treated as "not implemented yet", rather than actual limitations. However, some of them are implications of the design choices made. Those are clearly marked as such.

- BOOTP (RFC 951) is not supported. This is a design choice: BOOTP support is not planned.

- On Linux and BSD system families the DHCP messages are sent and received over the raw sockets (using LPF and BPF) and all packet headers (including data link layer, IP and UDP headers) are created and parsed by Kea, rather than 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.

Chapter 8

The DHCPv6 Server

8.1 Starting and Stopping the DHCPv6 Server

It is recommended that the Kea DHCPv6 server be started and stopped using **keactrl** (described in Chapter 6). However, it is also possible to run the server directly: it accepts the following command-line switches:

- **-c** *file* - specifies the configuration file. This is the only mandatory switch.

- **-d** - specifies whether the server logging should be switched to verbose mode. In verbose mode, the logging severity and debuglevel specified in the configuration file are ignored and "debug" severity and the maximum debuglevel (99) are assumed. The flag is convenient, for temporarily switching the server into maximum verbosity, e.g. when debugging.

- **-p** *port* - specifies UDP port on which the server will listen. This is only useful during testing, as a DHCPv6 server listening on ports other than the standard ones will not be able to handle regular DHCPv6 queries.

- **-v** - prints out the Kea version and exits.

- **-V** - prints out the Kea extended version with additional parameters and exits. The listing includes the versions of the libraries dynamically linked to Kea.

- **-W** - prints out the Kea configuration report and exits. The report is a copy of the `config.report` file produced by **./configure**: it is embedded in the executable binary.

The `config.report` may also be accessed more directly. The following command may be used to extract this information. The binary **path** may be found in the install directory or in the `.libs` subdirectory in the source tree. For example `kea/src/bin/dhcp6/.libs/kea-dhcp6`.

```
strings path/kea-dhcp6 | sed -n 's/;;;; //p'
```

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.

During startup the server will attempt to create a PID file of the form: localstatedir/[conf name].kea-dhcp6.pid where:

- **localstatedir**: The value as passed into the build configure script. It defaults to "/usr/local/var". Note that this value may be overridden at run time by setting the environment variable KEA_PIDFILE_DIR. This is intended primarily for testing purposes.

- **conf name**: The configuration file name used to start the server, minus all preceding path and file extension. For example, given a pathname of "/usr/local/etc/kea/myconf.txt", the portion used would be "myconf".

If the file already exists and contains the PID of a live process, the server will issue a DHCP6_ALREADY_RUNNING log message and exit. It is possible, though unlikely, that the file is a remnant of a system crash and the process to which the PID belongs is unrelated to Kea. In such a case it would be necessary to manually delete the PID file.

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

8.2 DHCPv6 Server Configuration

8.2.1 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 is as follows:

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

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

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

# And we specify the type of lease database
    "lease-database": {
        "type": "memfile",
        "persist": true,
        "name": "/var/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 together with their format. Subsequent sections of this chapter go into much greater detail for these and other parameters.

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

The configuration starts in the first line with the initial opening curly bracket (or brace). Each configuration 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 the 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 well formed JSON. That means that parameters for any given scope must be separated by a comma and there must not be a comma after the last parameter. When reordering a configuration file, keep in mind that moving a parameter to or from the last position in a given scope may also require moving the comma. The second caveat is that it is uncommon — although legal JSON — to repeat the same parameter multiple times. If that happens, the last occurrence of a given parameter in a given scope is used while all previous instances are ignored. This is unlikely to cause any confusion as there are no real life reasons to keep multiple copies of the same parameter in your configuration file.

Moving onto the DHCPv6 configuration elements, the very first few elements define some global parameters. **valid-lifetime** defines for how long the addresses (leases) given out by the server are valid. If nothing changes, a client that got an address is allowed to use it for 4000 seconds. (Note that integer numbers are specified as is, without any quotes around them.) The address will become deprecated in 3000 seconds (clients are allowed to keep old connections, but can't use this address for creating new connections). **renew-timer** and **rebind-timer** are values that define T1 and T2 timers that govern when the client will begin the renewal and rebind procedures.

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

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

The next couple of lines define the lease database, the place where the server stores its lease information. This particular example tells the server to use **memfile**, which is the simplest (and fastest) database backend. It uses an in-memory database and stores leases on disk in a CSV file. This is a very simple configuration. Usually the lease database configuration is more extensive and contains additional parameters. Note that **lease-database** is an object and opens up a new scope, using an opening brace. Its parameters (just one in this example - **type**) follow. Had there been more than one, they would be separated by commas. This scope is closed with a closing brace. As more parameters for the Dhcp6 definition follow, a trailing comma is present.

Finally, we need to define a list of IPv6 subnets. This is the most important DHCPv6 configuration structure as the server uses that information to process clients' requests. It defines all subnets from which the server is expected to receive DHCP requests. The subnets are specified with the **subnet6** parameter. It is a list, so it starts and ends with square brackets. Each subnet definition in the list has several attributes associated with it, so it is a structure and is opened and closed with braces. At 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 **subnet6** parameter would be specified and separated by commas. For example, to define two subnets, the following syntax would be used:

```
"subnet6": [
    {
        "pools": [ { "pool": "2001:db8:1::/112" } ],
        "subnet": "2001:db8:1::/64"
    },
    {
        "pools": [ { "pool": "2001:db8:2::1-2001:db8:2::ffff" } ],
        "subnet": "2001:db8:2::/64"
    }
]
```

Note that indentation is optional and is used for aesthetic purposes only. In some cases 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.

8.2.2 Lease Storage

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

8.2.2.1 Memfile - Basic Storage for Leases

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

The configuration of the file backend (Memfile) is controlled through the Dhcp6/lease-database parameters. The **type** parameter is mandatory and it specifies which storage for leases the server should use. The value of `"memfile"` indicates that the file should be used as the storage. The following list gives additional, optional, parameters that can be used to configure the Memfile backend.

- **persist**: controls whether the new leases and updates to existing leases are written to the file. It is strongly recommended that the value of this parameter is set to `true` at all times, during the server's normal operation. Not writing leases to disk 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. The value of `false` is mostly useful for performance testing purposes. The default value of the **persist** parameter is `true`, which enables writing lease updates to the lease file.

- **name**: specifies an absolute location of the lease file in which new leases and lease updates will be recorded. The default value for this parameter is `"[kea-install-dir]/var/kea/kea-leases6.csv"`.

- **lfc-interval**: specifies the interval in seconds, at which the server will perform a lease file cleanup (LFC). This removes redundant (historical) information from the lease file and effectively reduces the lease file size. The cleanup process is described in more detailed fashion further in this section. The default value of the **lfc-interval** is `0`, which disables the LFC.

An example configuration of the Memfile backend is presented below:

```
"Dhcp6": {
    "lease-database": {
        "type": "memfile",
        "persist": true,
        "name": "/tmp/kea-leases6.csv",
        "lfc-interval": 1800
    }
}
```

This configuration selects the `/tmp/kea-leases6.csv` as the storage for lease information and enables persistence (writing lease updates to this file). It also configures the backend perform the periodic cleanup of the lease files, executed every 30 minutes.

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

Lease file cleanup (LFC) removes all previous entries for each client and leaves only the latest ones. The interval at which the cleanup is performed is configurable, and it should be selected according to the frequency of lease renewals initiated by the clients. The more frequent the renewals, the smaller the value of **lfc-interval** should be. Note however, that the LFC takes time and thus it is possible (although unlikely) that new cleanup is started while the previous cleanup instance is still running, if

the **lfc-interval** is too short. The server would recover from this by skipping the new cleanup when it detects that the previous cleanup is still in progress. But it implies that the actual cleanups will be triggered more rarely than configured. Moreover, triggering a new cleanup adds an overhead to the server which will not be able to respond to new requests for a short period of time when the new cleanup process is spawned. Therefore, it is recommended that the **lfc-interval** value is selected in a way that would allow for the LFC to complete the cleanup before a new cleanup is triggered.

Lease file cleanup is performed by a separate process (in background) to avoid a performance impact on the server process. In order to avoid the conflicts between two processes both using the same lease files, the LFC process operates on the copy of the original lease file, rather than on the lease file used by the server to record lease updates. There are also other files being created as a side effect of the lease file cleanup. The detailed description of the LFC is located on the Kea wiki: http://kea.isc.org/wiki/LFCDesign.

8.2.2.2 Lease Database Configuration

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

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

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

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

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

If the database is located on a different system to the DHCPv6 server, the database host name must also be specified. (It should be noted that this configuration may have a severe impact on server performance.):

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

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" : "", ... }, ... }
```

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

```
"Dhcp6": { "lease-database": { "connect-timeout" : timeout-in-seconds, ... }, ... }
```

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

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.)

8.2.3 Hosts Storage

Kea is also able to store information about host reservations in the database. The hosts database configuration uses the same syntax as the lease database. In fact, a Kea server opens independent connections for each purpose, be it lease or hosts information. This arrangement gives the most flexibility. Kea can be used to keep leases and host reservations separately, but can also point to the same database. Currently the supported hosts database types are MySQL and PostgreSQL. The Cassandra backend does not support host reservations yet.

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

8.2.3.1 DHCPv6 Hosts Database Configuration

Hosts database configuration is controlled through the Dhcp6/hosts-database parameters. If enabled, the type of the database must be set to "mysql" or "postgresql". Other hosts backends may be added in later version of Kea.

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

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

```
"Dhcp6": { "hosts-database": { "name": "database-name" , ... }, ... }
```

If the database is located on a different system than the DHCPv6 server, the database host name must also be specified. (Again it should be noted that this configuration may have a severe impact on server performance):

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

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

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

```
"Dhcp6": { "hosts-database": { "user": "user-name",
                               "password": "password",
                               ... },
          ... }
```

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

8.2.3.2 Using Read-Only Databases for Host Reservations

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

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

```
"Dhcp6": { "hosts-database": { "readonly": true, ... }, ... }
```

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

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

8.2.4 Interface Selection

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

```
"Dhcp6": {
    "interfaces-config": {
        "interfaces": [ "*" ]
    }
    ...
}
```

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

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

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

```
"Dhcp6": {
    "interfaces-config": {
        "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.

8.2.5 IPv6 Subnet Identifier

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

If there are multiple subnets configured with auto-generated identifiers and one of them is removed, the subnet identifiers may be renumbered. For example: if there are four subnets and the third is removed the last subnet will be assigned the identifier that the third subnet had before removal. As a result, the leases stored in the lease database for subnet 3 are now associated with subnet 4, something that may have unexpected consequences. It is planned to implement a 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 specified subnet identifier to the newly configured subnet:

```
"Dhcp6": {
    "subnet6": [
        {
            "subnet": "2001:db8:1::/64",
            "id": 1024,
            ...
        }
    ]
}
```

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

8.2.6 Unicast Traffic Support

When the DHCPv6 server starts, by default it listens to the DHCP traffic sent to multicast address ff02::1:2 on each interface that it is configured to listen on (see Section 8.2.4). 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 this is to have relays send their traffic to the server directly. To configure the server to listen on a specific unicast address, nn interface name can be optionally followed by a slash, followed by the global unicast address on which the server should listen. The server listens to this address in addition to normal link-local binding and listening on ff02::1:2 address. The sample configuration below shows how to listen on 2001:db8::1 (a global address) configured on the eth1 interface.

```
"Dhcp6": {
    "interfaces-config": {
        "interfaces": [ "eth1/2001:db8::1" ]
    },
    ...
    "option-data": [
        {
            "name": "unicast",
            "data": "2001:db8::1"
        } ],
    ...
}
```

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

Usually unicast support is associated with a server unicast option which allows clients to send unicast messages to the server. The example above includes a server unicast option specification which will cause the client to send messages to the specified unicast address.

It is possible to mix interface names, wildcards and interface name/addresses in the list of interfaces. It is not possible however to specify more than one unicast address on a given interface.

Care should be taken to specify proper unicast addresses. The server will attempt to bind to the addresses specified without any additional checks. This approach has selected on purpose to allow the software to communicate over uncommon addresses if so desired.

8.2.7 Subnet and Address Pool

The main 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 **pools** parameter is actually a list of pools: for this reason, the pool definition is enclosed in square brackets, even though only one range of addresses is specified.

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

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

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

The server may be configured to serve more than one subnet. To add a second subnet, use a command similar to the following:

```
"Dhcp6": {
    "subnet6": [
    {
        "subnet": "2001:db8:1::/64",
        "pools": [
            { "pool": "2001:db8:1::1-2001:db8:1::ffff" }
        ]
    },
    {
        "subnet": "2001:db8:2::/64",
        "pools": [
            { "pool": "2001:db8:2::/64" }
        ]
    },

        ...
    ]
}
```

In this example, we allow the server to dynamically assign all addresses available in the whole subnet. Although rather wasteful, it is certainly a valid configuration to dedicate the whole /64 subnet for that purpose. Note that the Kea server does not preallocate the leases, so there is no danger in using gigantic address pools.

When configuring a DHCPv6 server using prefix/length notation, please pay attention to the boundary values. When specifying that the server can use a given pool, it will also be able to allocate the first (typically 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.

8.2.8 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 (**prefix**) and a prefix length (**prefix-len**), as well as a delegated prefix length (**delegated-len**). The delegated length must not be shorter (that is it must be numerically greater or equal) than the prefix length. If both the delegated and prefix lengths are equal, the server will be able to delegate only one prefix. The delegated prefix does not have to match the subnet prefix.

Below is a sample subnet configuration which enables prefix delegation for the subnet:

```
"Dhcp6": {
    "subnet6": [
        {
            "subnet": "2001:d8b:1::/64",
            "pd-pools": [
                {
                    "prefix": "3000:1::",
                    "prefix-len": 64,
                    "delegated-len": 96
                }
            ]
        }
    ],
    ...
}
```

8.2.9 Standard DHCPv6 Options

One of the major features of a 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 requests them. The following example shows how to configure DNS servers, one of the most frequently used options. Options specified in this way are considered global and apply to all configured subnets.

```
"Dhcp6": {
    "option-data": [
        {
            "name": "dns-servers",
            "code": 23,
            "space": "dhcp6",
            "csv-format": true,
            "data": "2001:db8::cafe, 2001:db8::babe"
        },
        ...
    ]
}
```

The **option-data** line creates a new entry in the option-data table. This table contains information on all global options that the server is supposed to configure in all subnets. The **name** line specifies the option name. (For a complete list of currently supported names, see Table 8.1.) The next line specifies the option code, which must match one of the values from that list. The

line beginning with **space** specifies the option space, which must always be set to "dhcp6" as these are standard DHCPv6 options. For other name spaces, including custom option spaces, see Section 8.2.12. The following line specifies the format in which the data will be entered: use of CSV (comma separated values) is recommended. Finally, the **data** line gives the actual value to be sent to clients. Data is specified as normal text, with values separated by commas if more than one value is allowed.

Options can also be configured as hexadecimal values. If "csv-format" is set to false, the option data must be specified as a 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"
        },
        ...
    ]
}
```

Note

The value for the setting of the "data" element is split across two lines in this example for clarity: when entering the command, the whole string should be entered on the same line.

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

Most of the parameters in the "option-data" structure are optional and can be omitted in some circumstances as discussed in the Section 8.2.13.

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

The currently supported standard DHCPv6 options are listed in Table 8.1. 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 7.3.

Experimental options (like standard options but with a code which was not assigned by IANA) are listed in Table 8.2.

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

The Section 8.2.10 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 yes provide a definition. In order to use such options, a server administrator must create a definition as described in Section 8.2.10 in the 'dhcp6' option space. This definition should match the option format described in the relevant RFC but the configuration mechanism would allow any option format as it has no means to validate the format at the moment.

8.2.10 Custom DHCPv6 Options

It is possible to define options in addition to the standard ones. Assume that we want to define a new DHCPv6 option called "foo" which will have code 100 and which will convey a single unsigned 32 bit integer value. We can define such an option by using the following commands:

```
"Dhcp6": {
    "option-def": [
        {
            "name": "foo",
            "code": 100,
            "type": "uint32",
            "array": false,
            "record-types": "",
            "space": "dhcp6",
            "encapsulate": ""
        }, ...
    ],
    ...
}
```

The "false" value of the **array** parameter determines that the option does NOT comprise an array of "uint32" values but 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-types** 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 example only defines the format of the new option, it does not set its value(s).

The **name**, **code** and **type** parameters are required, all others are optional. The **array** default value is **false**. The **record-types** and **encapsulate** default values are blank (i.e. ""). The default **space** is "dhcp6".

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

```
"Dhcp6": {
    "option-data": [
        {
            "name": "foo",
            "code": 100,
            "space": "dhcp6",
            "csv-format": true,
            "data": "12345"
        }, ...
    ],
```

Name	Code	Type	Array?
preference	7	uint8	false
unicast	12	ipv6-address	false
vendor-opts	17	uint32	false
sip-server-dns	21	fqdn	true
sip-server-addr	22	ipv6-address	true
dns-servers	23	ipv6-address	true
domain-search	24	fqdn	true
nis-servers	27	ipv6-address	true
nisp-servers	28	ipv6-address	true
nis-domain-name	29	fqdn	true
nisp-domain-name	30	fqdn	true
sntp-servers	31	ipv6-address	true
information-refresh-time	32	uint32	false
bcmcs-server-dns	33	fqdn	true
bcmcs-server-addr	34	ipv6-address	true
geoconf-civic	36	record (uint8, uint16, binary)	false
remote-id	37	record (uint32, binary)	false
subscriber-id	38	binary	false
client-fqdn	39	record (uint8, fqdn)	false
pana-agent	40	ipv6-address	true
new-posix-timezone	41	string	false
new-tzdb-timezone	42	string	false
ero	43	uint16	true
lq-query	44	record (uint8, ipv6-address)	false
client-data	45	empty	false
clt-time	46	uint32	false
lq-relay-data	47	record (ipv6-address, binary)	false
lq-client-link	48	ipv6-address	true
bootfile-url	59	string	false
bootfile-param	60	binary	false
client-arch-type	61	uint16	true
nii	62	record (uint8, uint8, uint8)	false
erp-local-domain-name	65	fqdn	false
rsoo	66	empty	false
client-linklayer-addr	79	binary	false
dhcp4o6-server-addr	88	ipv6-address	true

Table 8.1: List of Standard DHCPv6 Options

Name	Code	Type	Array?
public-key	701	binary	false
certificate	702	binary	false
signature	703	record (uint8, uint8, binary)	false
timestamp	704	binary	false

Table 8.2: List of Experimental DHCPv6 Options

```
        ...
}
```

New options can take more complex forms than simple use of primitives (uint8, string, ipv6-address etc): it is possible to define an option comprising a number of existing primitives.

For example, assume we want to define a new option that will consist of an IPv6 address, followed by an unsigned 16 bit integer, followed by a boolean value, followed by a text string. Such an option could be defined in the following way:

```
"Dhcp6": {
    "option-def": [
        {
            "name": "bar",
            "code": 101,
            "space": "dhcp6",
            "type": "record",
            "array": false,
            "record-types": "ipv6-address, uint16, boolean, string",
            "encapsulate": ""
        }, ...
    ],
    ...
}
```

The "type" is set to "record" to indicate that the option contains multiple values of different types. These types are given as a comma-separated list in the "record-types" field and should be those listed in Table 7.3.

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 the 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 versions of Kea will accept all those values for all boolean parameters.

8.2.11 DHCPv6 Vendor-Specific Options

Currently there are two option spaces defined for the DHCPv6 daemon: "dhcp6" (for top level DHCPv6 options) and "vendor-opts-space", which is empty by default, but in which options can be defined. Those options will be carried in the Vendor-Specific Information option (code 17). The following examples show how to define an option "foo" with code 1 that consists of an IPv6 address, an unsigned 16 bit integer and a string. The "foo" option is conveyed in a Vendor-Specific Information option. 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-opts-space",
            "type": "record",
            "array": false,
            "record-types": "ipv6-address, uint16, string",
            "encapsulate": ""
        }
    ],
    ...
}
```

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

```
"Dhcp6": {
    "option-data": [
        {
            "name": "foo",
            "space": "vendor-opts-space",
            "data": "2001:db8:1::10, 123, Hello World"
        },
        ...
    ],
    ...
}
```

We should also define a value (enterprise-number) for the Vendor-specific Information option, that conveys our option "foo".

```
"Dhcp6": {
    "option-data": [
        ...,
        {
            "name": "vendor-opts",
            "data": "12345"
        }
    ],
    ...
}
```

Alternatively, the option can be specified using its code.

```
"Dhcp6": {
    "option-data": [
        ...,
        {
            "code": 17,
            "data": "12345"
        }
    ],
    ...
}
```

8.2.12 Nested DHCPv6 Options (Custom Option Spaces)

It is sometimes useful to define completely new option spaces. This is useful if the user wants their new option to convey sub-options that use a separate numbering scheme, for example sub-options with codes 1 and 2. Those option codes conflict with standard DHCPv6 options, so a separate option space must be defined.

Note that it is not required to create a new option space when defining sub-options for a standard option because it is created by default if the standard option is meant to convey any sub-options (see Section 8.2.11).

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 limits this option to only carrying data in sub-options.

Finally, we can set values for the new options:

```
"Dhcp6": {
    "option-data": [
        {
            "name": "subopt1",
            "code": 1,
            "space": "isc",
            "data": "2001:db8::abcd"
        },
```

```
                }
                "name": "subopt2",
                "code": 2,
                "space": "isc",
                "data": "Hello world"
            },
            {

                "name": "container",
                "code": 102,
                "space": "dhcp6"
            }
        ],
        ...
}
```

Note that it is possible to create an option which carries some data in addition to the sub-options defined in the encapsulated option space. For example, if the "container" option from the previous example 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.

8.2.13 Unspecified Parameters for DHCPv6 Option Configuration

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

- **name** - the server requires an option name or option code to identify an option. If this parameter is unspecified, the option code must be specified.

- **code** - the server requires an option name or option code to identify an option. This parameter may be left unspecified if the **name** parameter is specified. However, this also requires that the particular option has its definition (it is either a standard option or an administrator created a definition for the option using an 'option-def' structure), as the option definition associates an option with a particular name. It is possible to configure an option for which there is no definition (unspecified option format). Configuration of such options requires the use of option code.

- **space** - if the option space is unspecified it will default to 'dhcp6' which is an option space holding DHCPv6 standard options.

- **data** - if the option data is unspecified it defaults to an empty value. The empty value is mostly used for the options which have no payload (boolean options), but it is legal to specify empty values for some options which carry variable length data and which spec allows for the length of 0. For such options, the data parameter may be omitted in the configuration.

- **csv-format** - if this value is not specified and the definition for the particular option exists, the server will assume that the option data is specified as a list of comma separated values to be assigned to individual fields of the DHCP option. If the definition does not exist for this option, the server will assume that the data parameter contains the option payload in the binary format (represented as a string of hexadecimal digits). Note that not specifying this parameter doesn't imply that it defaults to a fixed value, but the configuration data interpretation also depends on the presence of the option definition. An administrator must be aware if the definition for the particular option exists when this parameter is not specified. It is generally recommended to not specify this parameter only for the options for which the definition exists, e.g. standard options. Setting **csv-format** to an explicit value will cause the server to strictly check the format of the option data specified.

8.2.14 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 the server may have many subnet configurations defined, it must select an appropriate subnet for a given request.

The server can not assume which of the configured subnets are local. In IPv4 it is possible as there is a 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: the DHCPv6 server must be able to operate while only using link-local addresses. Therefore an optional **interface** parameter is available within a subnet definition to designate that a given subnet is local, i.e. reachable directly over the specified interface. For example the server that is intended to serve a local subnet over eth0 may be configured as follows:

```
"Dhcp6": {
    "subnet6": [
        {
            "subnet": "2001:db8:beef::/48",
            "pools": [
                {
                    "pool": "2001:db8:beef::/48"
                }
             ],
            "interface": "eth0"
        }
    ],
    ...
}
```

8.2.15 Rapid Commit

The Rapid Commit option, described in RFC 3315, is supported by the Kea DHCPv6 server. However, support is disabled by default for all subnets. It can be enabled for a particular subnet using the **rapid-commit** parameter as shown below:

```
"Dhcp6": {
    "subnet6": [
        {
            "subnet": "2001:db8:beef::/48",
            "rapid-commit": true,
            "pools": [
                {
                    "pool": "2001:db8:beef::1-2001:db8:beef::10"
                }
             ],
        }
    ],
    ...
}
```

This setting only affects the subnet for which the **rapid-commit** is set to **true**. For clients connected to other subnets, the server will ignore the Rapid Commit option sent by the client and will follow the 4-way exchange procedure, i.e. respond with an Advertise for a Solicit containing a Rapid Commit option.

8.2.16 DHCPv6 Relays

A DHCPv6 server with multiple subnets defined must select the appropriate subnet when it receives a request from a client. For clients connected via relays, two mechanisms are used:

The first uses the linkaddr field in the RELAY_FORW message. The name of this field is somewhat misleading in that it does not contain a link-layer address: instead, it holds an address (typically a global address) that is used to identify a link. The DHCPv6 server checks 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.

The following example configuration shows how to specify an interface-id with a value of "vlan123".

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

8.2.17 Relay-Supplied Options

RFC 6422 defines a mechanism called Relay-Supplied DHCP Options. In certain cases relay agents are the only entities that may have specific information. They can insert options when relaying messages from the client to the server. The server will then do certain checks and copy those options to the response that will be sent to the client.

There are certain conditions that must be met for the option to be included. First, the server must not provide the option itself. In other words, if both relay and server provide an option, the server always takes precedence. Second, the option must be RSOO-enabled. IANA maintains a list of RSOO-enabled options here. However, there may be cases when system administrators want to echo other options. Kea can be instructed to treat other options as RSOO-enabled. For example, to mark options 110, 120 and 130 as RSOO-enabled, the following syntax should be used:

```
"Dhcp6": {
    "relay-supplied-options": [ "110", "120", "130" ],
    ...
}
```

As of March 2015, only option 65 is RSOO-enabled by IANA. This option will always be treated as such and there's no need to explicitly mark it. Also, when enabling standard options, it is possible to use their names, rather than option code, e.g. (e.g. use **dns-servers** instead of **23**). See Table 8.1 for the names. In certain cases it could also work for custom options, but due to the nature of the parser code this may be unreliable and should be avoided.

8.2.18 Client Classification in DHCPv6

The DHCPv6 server includes support for client classification. For a deeper discussion of the classification process see Chapter 12.

In certain cases it is useful to differentiate between different types of clients and treat them accordingly. It is envisaged that client classification will be used for changing the behavior of almost any part of the DHCP message processing, including the assignment of leases from different pools, the assignment of different options (or different values of the same options) etc. In the current release of the software however, there are only two mechanisms that take advantage of client classification: subnet selection and assignment of different options.

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. Here, there are two classes of devices: the cable modem itself, which should be handed a lease from subnet A and all other devices behind the modem 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 12.6.

The process of doing classification is conducted in three steps. The first step is to assess an incoming packet and assign it to zero or more classes. The second step is to choose a subnet, possibly based on the class information. The third step is to assign options again possibly based on the class information.

There are two methods of doing classification. The first is automatic and relies on examining the values in the vendor class options. Information from these options is extracted and a class name is constructed from it and added to the class list for the packet. The second allows you to specify an expression that is evaluated for each packet. If the result is true the packet is a member of the class.

Note

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

8.2.18.1 Defining and Using Custom Classes

The following example shows how to configure a class using an expression and a subnet making use of that class. This configuration defines the class named "Client_enterprise". It is comprised of all clients whose client identifiers start with the given hex string (which would indicate a DUID based on an enterprise id of 0xAABBCCDD). They will be given an address from 2001:db8:1::0 to 2001:db8:1::FFFF and the addresses of their DNS servers set to 2001:db8:0::1 and 2001:db8:2::1.

```
"Dhcp6": {
    "client-classes": [
        {
            "name": "Client_enterprise",
            "test": "substring(option[1].hex,0,6) == 0x0002AABBCCDD'",
            "option-data": [
                {
                    "name": "dns-servers",
                    "code": 23,
                    "space": "dhcp6",
                    "csv-format": true,
                    "data": "2001:db8:0::1, 2001:db8:2::1"
                }
            ]
        },
        ...
    ],
    "subnet6": [
        {
            "subnet": "2001:db8:1::/64",
            "pools": [ { "pool": "2001:db8:1::-2001:db8:1::ffff" } ],
            "client-class": "Client_enterprise"
        }
    ],
    ...
}
```

This example shows a configuration using an automatically generated "VENDOR_CLASS_" class. The Administrator of the network has decided that addresses from range 2001:db8:1::1 to 2001:db8:1::ffff are going to be managed by the Dhcp6 server and only clients belonging to the eRouter1.0 client class are allowed to use that pool.

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

8.2.19 DDNS for DHCPv6

As mentioned earlier, kea-dhcp6 can be configured to generate requests to the DHCP-DDNS server (referred to here as "D2") to update DNS entries. These requests are known as 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 the kea-dhcp6 configuration. The mandatory parameters for the DHCP DDNS configuration are **enable-updates** which is unconditionally required, and **qualifying-suffix** which has no default value and is required when **enable-updates** is set to **true**. The two (disabled and enabled) minimal DHCP DDNS configurations are:

```
"Dhcp6": {
    "dhcp-ddns": {
        "enable-updates": false
    },
    ...
}
```

and for example:

```
"Dhcp6": {
    "dhcp-ddns": {
        "enable-updates": true,
        "qualifying-suffix": "example."
    },
    ...
}
```

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

- **"server-ip": "127.0.0.1"**

- **"server-port": 53001**

- **"sender-ip": ""**

- **"sender-port": 0**

- **"max-queue-size": 1024**

- **"ncr-protocol": "UDP"**

- **"ncr-format": "JSON"**

- **"override-no-update": false**

- **"override-client-update": false**

- **"replace-client-name": "never"**

- **"generated-prefix": "myhost"**

8.2.19.1 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 this communication:

- **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 a suitable port.

- **max-queue-size** - maximum number of requests allowed to queue waiting to be sent to D2. This value guards against requests accumulating uncontrollably if they are being generated faster than they can be delivered. If the number of requests queued for transmission reaches this value, DDNS updating will be turned off until the queue backlog has been sufficiently reduced. The intent is to allow kea-dhcp6 to continue lease operations. The default value is 1024.

- **ncr-protocol** - socket protocol use when sending requests to D2. Currently only UDP is supported. TCP may be available in an upcoming release.

- **ncr-format** - 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, kea-dhcp-ddns 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 configuration would be required:

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

8.2.19.2 When Does kea-dhcp6 Generate a DDNS Request?

kea-dhcp6 follows the behavior prescribed for DHCP servers in RFC 4704. It is important to keep in mind that kea-dhcp6 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 within the purview of D2 itself (Chapter 10). 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 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 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. When a new lease is granted, kea-dhcp6 will generate a DDNS update request only if the 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 8.3: Default FQDN Flag Behavior

The first row in the table above represents "client delegation". Here the DHCP client states that it intends to do the forward DNS updates and the server should do the reverse updates. By default, kea-dhcp6 will honor the client's wishes and generate a DDNS request to D2 to update only reverse DNS data. The parameter, **override-client-update**, can be used to instruct the server to override client delegation requests. When this parameter is true, kea-dhcp6 will disregard requests for client delegation and generate a DDNS request to update both forward and reverse DNS data. In this case, the N-S-O flags in the server's response to the client will be 0-1-1 respectively.

(Note that the flag combination N=1, S=1 is prohibited according to RFC 4702. If such a combination is received from the client, the packet will be dropped by kea-dhcp6.)

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

```
"Dhcp6": {
    "dhcp-ddns": {
        "override-client-update": true,
        ...
    },
    ...
}
```

The third row in the table above describes the case in which the client requests that no DNS updates be done. The parameter, **override-no-update**, can be used to instruct the server to disregard the client's wishes. When this parameter is true, kea-dhcp6 will generate DDNS update requests to kea-dhcp-ddns even if the client requests no updates be done. The N-S-O flags in the server's response to the client will be 0-1-1.

To override client delegation, issue the following commands:

```
"Dhcp6": {
    "dhcp-ddns": {
        "override-no-update": true,
        ...
    },
    ...
}
```

8.2.19.3 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.

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

1. If the DHCPREQUEST contains the client FQDN option, the candidate name is taken from there.

2. If the candidate name is a partial (i.e. unqualified) name then add a configurable suffix to the name and use the result as the FQDN.

3. If the candidate name provided is empty, generate an FQDN using a configurable prefix and suffix.

4. If the client provided neither option, then no DNS action will be taken.

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

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

- **always** - Replace the name the client sent. If the client sent no name, generate one for the client.

- **when-present** - Replace the name the client sent. If the client sent no name, do not generate one.

- **when-not-present** - Use the name the client sent. If the client sent no name, generate one for the client.

Note Note that formerly, this parameter was a boolean and permitted only values of **true** and **false**. Boolean values will still be accepted but may eventually be deprecated. A value of **true** equates to **when-present**, **false** equates to **never**.

For example, To instruct kea-dhcp6 to always generate the FQDN for a client, set the parameter **replace-client-name** to **always** as follows:

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

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

```
"Dhcp6": {
    "dhcp-ddns": {
        "generated-prefix": "another.host",
        ...
    },
    ...
}
```

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

```
"Dhcp6": {
    "dhcp-ddns": {
        "qualifying-suffix": "foo.example.org",
        ...
    },
    ...
}
```

When qualifying a partial name, kea-dhcp6 will construct 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 qualifying-suffix "example.com", the generated FQDN would be:

some-computer.example.com.

When generating 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, the qualifying suffix "example.com", and the default value is used for **generated-prefix**, the generated FQDN would be:

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

8.2.20 DHCPv4-over-DHCPv6: DHCPv6 Side

The support of DHCPv4-over-DHCPv6 transport is described in RFC 7341 and is implemented using cooperating DHCPv4 and DHCPv6 servers. This section is about the configuration of the DHCPv6 side (the DHCPv4 side is described in Section 7.2.19).

Note DHCPv4-over-DHCPv6 support is experimental and the details of the inter-process communication can change: both the DHCPv4 and DHCPv6 sides should be running the same version of Kea.

There is only one specific parameter for the DHCPv6 side: **dhcp4o6-port** which specifies the first of the two consecutive ports of the UDP sockets used for the communication between the DHCPv6 and DHCPv4 servers (the DHCPv6 server is bound to ::1 on **port** and connected to ::1 on **port** + 1).

Two other configuration entries are in general required: unicast traffic support (see Section 8.2.6) and DHCP 4o6 server address option (name "dhcp4o6-server-addr", code 88).

The following configuration was used during some tests:

```
{

# DHCPv6 conf
"Dhcp6": {
```

```
    "interfaces-config": {
        "interfaces": [ "eno33554984/2001:db8:1:1::1" ]
    },

    "lease-database": {
        "type": "memfile",
        "name": "leases6"
    },

    "preferred-lifetime": 3000,
    "valid-lifetime": 4000,
    "renew-timer": 1000,
    "rebind-timer": 2000,

    "subnet6": [ {
        "subnet": "2001:db8:1:1::/64",
        "interface": "eno33554984",
        "pools": [ { "pool": "2001:db8:1:1::1:0/112" } ]
    } ],

    "dhcp4o6-port": 6767,

    "option-data": [ {
        "name": "dhcp4o6-server-addr",
        "code": 88,
        "space": "dhcp6",
        "csv-format": true,
        "data": "2001:db8:1:1::1"
    } ]

},

"Logging": {
    "loggers": [ {
        "name": "kea-dhcp6",
        "output_options": [ {
            "output": "/tmp/kea-dhcp6.log"
        } ],
        "severity": "DEBUG",
        "debuglevel": 0
    } ]
}

}
```

Note Relayed DHCPv4-QUERY DHCPv6 messages are not yet supported.

8.3 Host Reservation in DHCPv6

There are many cases where it is useful to provide a configuration on a per host basis. The most obvious one is to reserve specific, static IPv6 address or/and prefix for exclusive use by a given client (host) - returning client will get the same address or/and prefix every time and other clients will never get that address. Note that there may be cases when the new reservation has been made for the client for the address or prefix being currently in use by another client. We call this situation a "conflict". The conflicts get resolved automatically over time as described in the subsequent sections. Once conflict is resolved, the client will keep receiving the reserved configuration when it renews.

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

Hosts reservations are defined as parameters for each subnet. Each host can be identified by either DUID or its hardware/MAC address. See Section 8.9 for details. There is an optional **reservations** array in the **subnet6** structure. Each element in that array is a structure, that holds information about a single host. In particular, the structure has an identifier that uniquely identifies a host. In the DHCPv6 context, such an identifier is usually a DUID, but can also be a hardware or MAC address. Also, either one or more addresses or prefixes may be specified. It is possible to specify a hostname and DHCPv6 options for a given host.

The following example shows how to reserve addresses and prefixes for specific hosts:

```
"subnet6": [
    {
        "subnet": "2001:db8:1::/48",
        "pools": [ { "pool": "2001:db8:1::/80" } ],
        "pd-pools": [
            {
                "prefix": "2001:db8:1:8000::",
                "prefix-len": 56,
                "delegated-len": 64
            }
        ],
        "reservations": [
            {
                "duid": "01:02:03:04:05:0A:0B:0C:0D:0E",
                "ip-addresses": [ "2001:db8:1::100" ]
            },
            {
                "hw-address": "00:01:02:03:04:05",
                "ip-addresses": [ "2001:db8:1::101, 2001:db8:1::102" ]
            },
            {
                "duid": "01:02:03:04:05:06:07:08:09:0A",
                "ip-addresses": [ "2001:db8:1::103" ],
                "prefixes": [ "2001:db8:2:abcd::/64" ],
                "hostname": "foo.example.com"
            }
        ]
    }
]
```

This example includes reservations for three different clients. The first reservation is made for the address 2001:db8:1::100 for a client using DUID 01:02:03:04:05:0A:0B:0C:0D:0E. The second reservation is made for two addresses 2001:db8:1::101 and 2001:db8:1::102 for a client using MAC address 00:01:02:03:04:05. Lastly, address 2001:db8:1::103 and prefix 2001:db8:2:abcd::/64 are reserved for a client using DUID 01:02:03:04:05:06:07:08:09:0A. The last reservation also assigns a hostname to this client.

Note that DHCPv6 allows for a single client to lease multiple addresses and multiple prefixes at the same time. Therefore **ip-addresses** and **prefixes** are plural and are actually arrays. When the client sends multiple IA options (IA_NA or IA_PD), each reserved address or prefix is assigned to an individual IA of the appropriate type. If the number of IAs of specific type is lower than the number of reservations of that type, the number of reserved addresses or prefixes assigned to the client is equal to the number of IA_NAs or IA_PDs sent by the client, i.e. some reserved addresses or prefixes are not assigned. However, they still remain reserved for this client and the server will not assign them to any other client. If the number of IAs of specific type sent by the client is greater than the number of reserved addresses or prefixes, the server will try to assign all reserved addresses or prefixes to the individual IAs and dynamically allocate addresses or prefixes to remaining IAs. If the server cannot assign a reserved address or prefix because it is in use, the server will select the next reserved address or prefix and try to assign it to the client. If the server subsequently finds that there are no more reservations that can be assigned to the client at the moment, the server will try to assign leases dynamically.

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

include only one identifier, either DUID or hardware address. Defining both for the same host is considered a configuration error, but as of 1.1.0, it is not rejected.

Adding host reservation incurs a performance penalty. In principle, when a server that does not support host reservation responds to a query, it needs to check whether there is a lease for a given address being considered for allocation or renewal. The server that also supports host reservation, has to perform additional checks: not only if the address is currently used (i.e. if there is a lease for it), but also whether the address could be used by someone else (i.e. if there is a reservation for it). That additional check incurs additional overhead.

8.3.1 Address/Prefix Reservation Types

In a typical scenario there is an IPv6 subnet defined with a certain part of it dedicated for dynamic address allocation by the DHCPv6 server. There may be an additional address space defined for prefix delegation. Those dynamic parts are referred to as dynamic pools, address and prefix pools or simply pools. In principle, the host reservation can reserve any address or prefix that belongs to the subnet. The reservations that specify an address that belongs to configured pools are called "in-pool reservations". In contrast, those that do not belong to dynamic pools are called "out-of-pool reservations". There is no formal difference in the reservation syntax and both reservation types are handled uniformly. However, upcoming releases may offer improved performance if there are only out-of-pool reservations as the server will be able to skip reservation checks when dealing with existing leases. Therefore, system administrators are encouraged to use out-of-pool reservations if possible.

8.3.2 Conflicts in DHCPv6 Reservations

As reservations and lease information are stored separately, conflicts may arise. Consider the following series of events. The server has configured the dynamic pool of addresses from the range of 2001:db8::10 to 2001:db8::20. Host A requests an address and gets 2001:db8::10. Now the system administrator decides to reserve address 2001:db8::10 for Host B. In general, reserving an address that is currently assigned to someone else is not recommended, but there are valid use cases where such an operation is warranted.

The server now has a conflict to resolve. Let's analyze the situation here. If Host B boots up and request an address, the server is not able to assign the reserved address 2001:db8::10. A naive approach would to be immediately remove the lease for Host A and create a new one for Host B. That would not solve the problem, though, because as soon as Host B get the address, it will detect that the address is already in use by someone else (Host A) and would send a Decline message. Therefore in this situation, the server has to temporarily assign a different address from the dynamic pool (not matching what has been reserved) to Host B.

When Host A renews its address, the server will discover that the address being renewed is now reserved for someone else (Host B). Therefore the server will remove the lease for 2001:db8::10, select a new address and create a new lease for it. It will send two addresses in its response: the old address with lifetime set to 0 to explicitly indicate that it is no longer valid and the new address with a non-zero lifetime. When Host B renews its temporarily assigned address, the server will detect that the existing lease does not match reservation, so it will release the current address Host B has and will create a new lease matching the reservation. Similar as before, the server will send two addresses: the temporarily assigned one with zeroed lifetimes, and the new one that matches reservation with proper lifetimes set.

This recovery will succeed, even if other hosts will attempt to get the reserved address. Had Host C requested address 2001:db8::10 after the reservation was made, the server will propose a different address.

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

8.3.3 Reserving a Hostname

When the reservation for the client includes the **hostname**, the server will assign this hostname to the client and send it back in the Client FQDN, if the client sent the FQDN option to the server. The reserved hostname always takes precedence over the hostname supplied by the client (via the FQDN option) or the autogenerated (from the IPv6 address) hostname.

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

```
"subnet6": [
    {
        "subnet": "2001:db8:1::/48",
        "pools": [ { "pool": "2001:db8:1::/80" } ],
        "reservations": [
            {
                "duid": "01:02:03:04:05:0A:0B:0C:0D:0E",
                "ip-addresses": [ "2001:db8:1::100" ]
                "hostname": "alice-laptop"
            }
        ]
    }
],
"dhcp-ddns": {
    "enable-updates": true,
    "qualifying-suffix": "example.isc.org."
}
```

will result in assigning the "alice-laptop.example.isc.org." hostname to the client using the DUID "01:02:03:04:05:0A:0B:0C:0D:0E".
If the **qualifying-suffix** is not specified, the default (empty) value will be used, and in this case the value specified as a **hostname**
will be treated as fully qualified name. Thus, by leaving the **qualifying-suffix** empty it is possible to qualify hostnames for the
different clients with different domain names:

```
"subnet6": [
    {
        "subnet": "2001:db8:1::/48",
        "pools": [ { "pool": "2001:db8:1::/80" } ],
        "reservations": [
            {
                "duid": "01:02:03:04:05:0A:0B:0C:0D:0E",
                "ip-addresses": [ "2001:db8:1::100" ]
                "hostname": "mark-desktop.example.org."
            }
        ]
    }
],
"dhcp-ddns": {
    "enable-updates": true,
}
```

The above example results in the assignment of the "mark-desktop.example.org." hostname to the client using the DUID "01:02:03:04:05:

8.3.4 Including Specific DHCPv6 Options in Reservations

Kea 1.1.0 introduced the ability to specify options on a per host basis. The options follow the same rules as any other op-
tions. These can be standard options (see Section 8.2.9), custom options (see Section 8.2.10) or vendor specific options (see
Section 8.2.11). The following example demonstrates how standard options can be defined.

```
"reservations": [
{
   "duid": "01:02:03:05:06:07:08",
   "ip-addresses": [ "2001:db8:1::2" ],
    "option-data": [
    {
        "option-data": [ {
            "name": "dns-servers",
            "data": "3000:1::234"
        },
        {
```

```
            "name": "nis-servers",
            "data": "3000:1::234"
        }
    } ]
} ]
```

Vendor specific options can be reserved in a similar manner:

```
"reservations": [
{
    "duid": "aa:bb:cc:dd:ee:ff",
    "ip-addresses": [ "2001:db8::1" ],
    "option-data": [
    {
        "name": "vendor-opts",
        "data": 4491"
    },
    {
        "name": "tftp-servers",
        "space": "vendor-4491",
        "data": "3000:1::234"
    } ]
} ]
```

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

8.3.5 Reserving Client Classes in DHCPv6

The Section 12.4 explains how to configure the server to assign classes to a client based on the content of the options that this client sends to the server. Host reservations mechanisms also allow for the static assignment of classes to clients. The definitions of these classes are placed in the Kea configuration. The following configuration snippet shows how to specify that the client belongs to classes **reserved-class1** and **reserved-class2**. Those classes are associated with specific options being sent to the clients which belong to them.

```
{
    "client-classes": [
    {
      "name": "reserved-class1",
      "option-data": [
      {
          "name": "dns-servers",
          "data": "2001:db8:1::50"
      }
      ]
   },
   {
      "name": "reserved-class2",
      "option-data": [
      {
          "name": "nis-servers",
          "data": "2001:db8:1::100"
      }
      ]
    }
    ],
    "subnet6": [
    {   "pools": [ { "pool": "2001:db8:1::/64" } ],
        "subnet": "2001:db8:1::/48",
        "reservations": [
```

```
        {
            "duid": "01:02:03:04:05:06:07:08",

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

        } ]
    } ]
}
```

Static class assignments, as shown above, can be used in conjuction with classification using expressions.

8.3.6 Storing Host Reservations in MySQL or PostgreSQL

It is possible to store host reservations in MySQL or PostgreSQL. See Section 8.2.3 for information on how to configure Kea to use reservations stored in MySQL or PostgreSQL. Kea does not provide any dedicated tools for managing reservations in a database. The Kea wiki http://kea.isc.org/wiki/HostReservationsHowTo provides detailed information and examples of how reservations can be inserted into the database.

Note
In Kea 1.1.0 maximum length of an option specified per host is arbitrarily set to 4096 bytes.

8.3.7 Storing Host Reservations in CQL (Cassandra)

Kea currently does not support storing reservations in Cassandra (CQL).

8.3.8 Fine Tuning DHCPv6 Host Reservation

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

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

- **all** - enables all host reservation types. This is the default value. This setting is the safest and the most flexible. It allows in-pool and out-of-pool reservations. As all checks are conducted, it is also the slowest.

- **out-of-pool** - allows only out of pool host reservations. With this setting in place, the server may assume that all host reservations are for addresses that do not belong to the dynamic pool. Therefore it can skip the reservation checks when dealing with in-pool addresses, thus improving performance. Do not use this mode if any of your reservations use in-pool address. Caution is advised when using this setting. Kea 1.1.0 does not sanity check the reservations against **reservation-mode** and misconfiguration may cause problems.

- **disabled** - host reservation support is disabled. As there are no reservations, the server will skip all checks. Any reservations defined will be completely ignored. As the checks are skipped, the server may operate faster in this mode.

An example configuration that disables reservation looks like follows:

```
"Dhcp6": {
    "subnet6": [
        {
        "subnet": "2001:db8:1::/64",
        "reservation-mode": "disabled",
        ...
        }
    ]
}
```

Another aspect of the host reservations are different types of identifiers. Kea 1.1.0 supports two types of identifiers in DHCPv6: hw-address and duid, but more identifier types are likely to be added in the future. This is beneficial from a usability perspective. However, there is a drawback. For each incoming packet Kea has to to extract each identifier type and then query the database to see if there is a reservation done by this particular identifier. If nothing is found, the next identifier is extracted and next query is issued. This process continues until either a reservation is found or all identifier types have been checked. Over time with an increasing number of supported identifier types, Kea would become slower and slower.

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

```
"host-reservation-identifiers": [ "duid", "hw-address" ],
"subnet6": [
    {
        "subnet": "2001:db8:1::/64",
        ...
    }
]
```

If not specified, the default value is:

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

8.4   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. RFC 3315 defines three DUID types: DUID-LLT, DUID-EN and DUID-LL. RFC 6355 also defines DUID-UUID. Future specifications may introduce new DUID types.

The Kea DHCPv6 server generates a server identifier once, upon the first startup, and stores it in a file. This identifier isn't modified across restarts of the server and so is a stable identifier.

Kea follows recommendation from RFC 3315 to use DUID-LLT as the default server identifier. However, we have received reports that some deployments require different DUID types, and there is a need to administratively select both DUID type and/or its contents.

The server identifier can be configured using parameters within the **server-id** map element in the global scope of the Kea configuration file. The following example demonstrates how to select DUID-EN as a server identifier:

```
"Dhcp6": {
    "server-id": {
        "type": "EN"
    },
    ...
}
```

Currently supported values for **type** parameter are: "LLT", "EN" and "LL", for DUID-LLT, DUID-EN and DUID-LL respectively.

When a new DUID type is selected the server will generate its value and replace any existing DUID in the file. The server will then use the new server identifier in all future interactions with the clients.

> **Note**
> If the new server identifier is created after some clients have obtained their leases, the clients using the old identifier will not be able to renew the leases: the server will ignore messages containing the old server identifier. Clients will continue sending Renew until they transition to the rebinding state. In this state they will start sending Rebind messages to multicast address without a server identifier. The server will respond to the Rebind messages with a new server identifier and the clients will associate the new server identifier with their leases. Although the clients will be able to keep their leases and will eventually learn the new server identifier, this will be at the cost of increased number of renewals and multicast traffic due to a need to rebind. Therefore it is recommended that modification of the server identifier type and value is avoided if the server has already assigned leases and these leases are still valid.

There are cases when an administrator needs to explicitly specify a DUID value rather than allow the server to generate it. The following example demonstrates how to explicitly set all components of a DUID-LLT.

```
"Dhcp6": {
    "server-id": {
        "type": "LLT",
        "htype": 8,
        "identifier": "A65DC7410F05",
        "time": 2518920166
    },
    ...
}
```

where:

- **htype** is a 16-bit unsigned value specifying hardware type,

- **identifier** is a link layer address, specified as a string of hexadecimal digits,

- **time** is a 32-bit unsigned time value.

The hexadecimal representation of the DUID generated as a result of the configuration specified above will be:

```
 00:01:00:08:96:23:AB:E6:A6:5D:C7:41:0F:05
|type |htype|    time    |    identifier    |
```

It is allowed to use special value of 0 for "htype" and "time", which indicates that the server should use ANY value for these components. If the server already uses a DUID-LLT it will use the values from this DUID. If the server uses a DUID of a different type or doesn't use any DUID yet, it will generate these values. Similarly, if the "identifier" is assigned an empty string, the value of the identifier will be generated. Omitting any of these parameters is equivalent to setting them to those special values.

For example, the following configuration:

```
"Dhcp6": {
    "server-id": {
        "type": "LLT",
        "htype": 0,
        "identifier": "",
        "time": 2518920166
    },
    ...
}
```

indicates that the server should use ANY link layer address and hardware type. If the server is already using DUID-LLT it will use the link layer address and hardware type from the existing DUID. If the server is not using any DUID yet, it will use link layer address and hardware type from one of the available network interfaces. The server will use an explicit value of time. If it is different than a time value present in the currently used DUID, that value will be replaced, effectively causing modification of the current server identifier.

The following example demonstrates an explicit configuration of a DUID-EN:

```
"Dhcp6": {
    "server-id": {
        "type": "EN",
        "enterprise-id": 2495,
        "identifier": "87ABEF7A5BB545"
    },
    ...
}
```

where:

- **enterprise-id** is a 32-bit unsigned value holding enterprise number,

- **identifier** is a variable length identifier within DUID-EN.

The hexadecimal representation of the DUID-EN created according to the configuration above is:

```
 00:02:00:00:09:BF:87:AB:EF:7A:5B:B5:45
|type |  ent-id  |     identifier     |
```

As in the case of the DUID-LLT, special values can be used for the configuration of the DUID-EN. If **enterprise-id** is 0, the server will use a value from the existing DUID-EN. If the server is not using any DUID or the existing DUID has a different type, the ISC enterprise id will be used. When an empty string is used for **identifier**, the identifier from the existing DUID-EN will be used. If the server is not using any DUID-EN the new 6-bytes long identifier will be generated.

DUID-LL is configured in the same way as DUID-LLT with an exception that the **time** parameter has no effect for DUID-LL, because this DUID type only comprises a hardware type and link layer address. The following example demonstrates how to configure DUID-LL:

```
"Dhcp6": {
    "server-id": {
        "type": "LL",
        "htype": 8,
        "identifier": "A65DC7410F05"
    },
    ...
}
```

which will result in the following server identifier:

```
 00:03:00:08:A6:5D:C7:41:0F:05
|type |htype|   identifier    |
```

The server stores the generated server identifier in the following location: [kea-install-dir]/var/kea/kea-dhcp6-serverid.

In some uncommon deployments where no stable storage is available, the server should be configured not to try to store the server identifier. This choice is controlled by the value of **persist** boolean parameter:

```
"Dhcp6": {
    "server-id": {
        "type": "EN",
        "enterprise-id": 2495,
        "identifier": "87ABEF7A5BB545",
        "persist": false
```

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

The default value of the "persist" parameter is **true** which configures the server to store the server identifier on a disk.

In the example above, the server is configured to not store the generated server identifier on a disk. But, if the server identifier is not modified in the configuration the same value will be used after server restart, because entire server identifier is explicitly specified in the configuration.

8.5 Stateless DHCPv6 (Information-Request Message)

Typically DHCPv6 is used to assign both addresses and options. These assignments (leases) have state that changes over time, hence their name, stateful. DHCPv6 also supports a stateless mode, where clients request configuration options only. This mode is considered lightweight from the server perspective as it does not require any state tracking; hence its name.

The Kea server supports stateless mode. Clients can send Information-Request messages and the server will send back answers with the requested options (providing the options are available in the server configuration). The server will attempt to use per-subnet options first. If that fails - for whatever reason - it will then try to provide options defined in the global scope.

Stateless and stateful mode can be used together. No special configuration directives are required to handle this. Simply use the configuration for stateful clients and the stateless clients will get just options they requested.

This usage of global options allows for an interesting case. It is possible to run a server that provides just options and no addresses or prefixes. If the options have the same value in each subnet, the configuration can define required options in the global scope and skip subnet definitions altogether. Here's a simple example of such a configuration:

```
"Dhcp6": {
    "interfaces-config": {
        "interfaces": [ "ethX" ]
    },
    "option-data": [ {
        "name": "dns-servers",
        "data": "2001:db8::1, 2001:db8::2"
    } ],
    "lease-database": { "type": "memfile" }
}
```

This very simple configuration will provide DNS server information to all clients in the network, regardless of their location. Note the specification of the memfile lease database: this is needed as Kea requires a lease database to be specified even if it is not used.

8.6 Support for RFC 7550

The RFC 7550 introduced some changes to the DHCPv6 protocol to resolve a few issues with the coexistence of multiple stateful options in the messages sent between the clients and servers.

The typical example is when the client, such as a requesting router, requests an allocation of both addresses and prefixes when it performs the 4-way (SARR) exchange with the server. If the server is not configured to allocate any prefixes but it can allocate some addresses, it will respond with the IA_NA(s) containing allocated addresses and the IA_PD(s) containing the NoPrefixAvail status code. If the client can operate without prefixes it may transition to the 'bound' state when it sends Renew/Rebind messages to the server, according to the T1 and T2 times, to extend the lifetimes of the allocated addresses. If the client is still interested in obtaining prefixes from the server it may also include an IA_PD in the Renew/Rebind to request allocation of the prefixes. If the server still cannot allocate the prefixes, it will respond with the IA_PD(s) containing NoPrefixAvail status code. However, if the server can now allocate the prefixes it will do so, and send them in the IA_PD(s) to the client. Allocation of leases during the Renew/Rebind was not supported in the RFC 3315 and RFC 3633, and has been introduced in RFC 7550. Kea supports this new behavior and it doesn't provide any configuration mechanisms to disable it.

The following are the other behaviors specified in the RFC 7550 supported by the Kea DHCPv6 server:

- Set T1/T2 timers to the same value for all stateful (IA_NA and IA_PD) options to facilitate renewal of all client's leases at the same time (in a single message exchange),

- NoAddrsAvail and NoPrefixAvail status codes are placed in the IA_NA and IA_PD options in the Advertise message, rather than as the top level options.

8.7 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 the interface that belongs to the subnet from which the server will assign addresses. In the typical case, the server is able to use the IPv6 address inserted by the relay (in the link-addr field in RELAY-FORW message) to select the appropriate subnet.

However, that is not always the case. The relay address may not match the subnet in certain deployments. This usually means that there is more than one subnet allocated for a given link. The two most common examples where this is the case are long lasting network renumbering (where both old and new address space is still being used) and a cable network. In a cable network both cable modems and the devices behind them are physically connected to the same link, yet they use distinct addressing. In such 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 a 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 with 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"
            }
        }
    ]
}
```

8.8 Segregating IPv6 Clients in a Cable Network

In certain cases, it is useful to mix relay address information, introduced in Section 8.7 with client classification, explained in Chapter 12. One specific example is a 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"
        }
    }
    ]
}
```

8.9 MAC/Hardware Addresses in DHCPv6

MAC/hardware addresses are available in DHCPv4 messages from the clients and administrators frequently use that information to perform certain tasks, like per host configuration, address reservation for specific MAC addresses and other. Unfortunately, the DHCPv6 protocol does not provide any completely reliable way to retrieve that information. To mitigate that issue a number of mechanisms have been implemented in Kea that attempt to gather it. Each of those mechanisms works in certain cases, but may fail in other cases. Whether the mechanism works or not in the particular deployment is somewhat dependent on the network topology and the technologies used.

Kea allows configuration of which of the supported methods should be used and in what order. This configuration may be considered a fine tuning of the DHCP deployment. In a typical deployment the default value of **"any"** is sufficient and there is no need to select specific methods. Changing the value of this parameter is the most useful in cases when an administrator wants to disable certain method, e.g. if the administrator trusts the network infrastructure more than the information provided by the clients themselves, the administrator may prefer information provided by the relays over that provided by the clients.

The configuration is controlled by the **mac-sources** parameter as follows:

```
"Dhcp6": {
    "mac-sources": [ "method1", "method2", "method3", ... ],

    "subnet6": [ ... ],

    ...
}
```

When not specified, a special value of "any" is used, which instructs the server to attempt to use all the methods in sequence and use value returned by the first one that succeeds.

Supported methods are:

- **any** - Not an actual method, just a keyword that instructs Kea to try all other methods and use the first one that succeeds. This is the default operation if no **mac-sources** are defined.

- **raw** - In principle, a DHCPv6 server could use raw sockets to receive incoming traffic and extract MAC/hardware address information. This is currently not implemented for DHCPv6 and this value has no effect.

- **duid** - DHCPv6 uses DUID identifiers instead of MAC addresses. There are currently four DUID types defined, with two of them (DUID-LLT, which is the default one and DUID-LL) convey MAC address information. Although RFC 3315 forbids it,

it is possible to parse those DUIDs and extract necessary information from them. This method is not completely reliable, as clients may use other DUID types, namely DUID-EN or DUID-UUID.

- **ipv6-link-local** - Another possible acquisition method comes from the source IPv6 address. In typical usage, clients are sending their packets from IPv6 link-local addresses. There is a good chance that those addresses are based on EUI-64, which contains MAC address. This method is not completely reliable, as clients may use other link-local address types. In particular, privacy extensions, defined in RFC 4941, do not use MAC addresses. Also note that successful extraction requires that the address's u-bit must be set to 1 and its g-bit set to 0, indicating that it is an interface identifier as per RFC 2373, section 2.5.1.

- **client-link-addr-option** - One extension defined to alleviate missing MAC issues is client link-layer address option, defined in RFC 6939. This is an option that is inserted by a relay and contains information about client's MAC address. This method requires a relay agent that supports the option and is configured to insert it. This method is useless for directly connected clients. This parameter can also be specified as **rfc6939**, which is an alias for **client-link-addr-option**.

- **remote-id** - RFC 4649 defines a remote-id option that is inserted by a relay agent. Depending on the relay agent configuration, the inserted option may convey the client's MAC address information. This parameter can also be specified as **rfc4649**, which is an alias for **remote-id**.

- **subscriber-id** - Another option that is somewhat similar to the previous one is subscriber-id, defined in RFC 4580. It is, too, inserted by a relay agent that is configured to insert it. This parameter can also be specified as **rfc4580**, which is an alias for **subscriber-id**. This method is currently not implemented.

- **docsis-cmts** - Yet another possible source of MAC address information are the DOCSIS options inserted by a CMTS that acts as a DHCPv6 relay agent in cable networks. This method attempts to extract MAC address information from suboption 1026 (cm mac) of the vendor specific option with vendor-id=4491. This vendor option is extracted from the relay-forward message, not the original client's message.

- **docsis-modem** - Yet another possible source of MAC address information are the DOCSIS options inserted by the cable modem itself. This method attempts to extract MAC address information from suboption 36 (device id) of the vendor specific option with vendor-id=4491. This vendor option is extracted from the original client's message, not from any relay options.

8.10 Duplicate Addresses (DECLINE Support)

The DHCPv6 server is configured with a certain pool of addresses that it is expected to hand out to the DHCPv6 clients. It is assumed that the server is authoritative and has complete jurisdiction over those addresses. However, due to various reasons, such as misconfiguration or a faulty client implementation that retains its address beyond the valid lifetime, there may be devices connected that use those addresses without the server's approval or knowledge.

Such an unwelcome event can be detected by legitimate clients (using Duplicate Address Detection) and reported to the DHCPv6 server using a DECLINE message. The server will do a sanity check (if the client declining an address really was supposed to use it), then will conduct a clean up operation and confirm it by sending back a REPLY message. Any DNS entries related to that address will be removed, the fact will be logged and hooks will be triggered. After that is done, the address will be marked as declined (which indicates that it is used by an unknown entity and thus not available for assignment to anyone) and a probation time will be set on it. Unless otherwise configured, the probation period lasts 24 hours. After that period, the server will recover the lease (i.e. put it back into the available state) and the address will be available for assignment again. It should be noted that if the underlying issue of a misconfigured device is not resolved, the duplicate address scenario will repeat. On the other hand, it provides an opportunity to recover from such an event automatically, without any sysadmin intervention.

To configure the decline probation period to a value other than the default, the following syntax can be used:

```
  "Dhcp6": {
    "decline-probation-period": 3600,
    "subnet6": [ ... ],
    ...
}
```

The parameter is expressed in seconds, so the example above will instruct the server to recycle declined leases after an hour.

There are several statistics and hook points associated with the Decline handling procedure. The lease6_decline hook is triggered after the incoming Decline message has been sanitized and the server is about to decline the lease. The declined-addresses

statistic is increased after the hook returns (both global and subnet specific variants). (See Section 7.7 and Chapter 13 for more details on DHCPv4 statistics and Kea hook points.)

Once the probation time elapses, the declined lease is recovered using the standard expired lease reclamation procedure, with several additional steps. In particular, both declined-addresses statistics (global and subnet specific) are decreased. At the same time, reclaimed-declined-addresses statistics (again in two variants, global and subnet specific) are increased.

Note about statistics: The server does not decrease the assigned-addresses statistics when a DECLINE message is received and processed successfully. While technically a declined address is no longer assigned, the primary usage of the assigned-addresses statistic is to monitor pool utilization. Most people would forget to include declined-addresses in the calculation, and simply do assigned-addresses/total-addresses. This would have a bias towards under-representing pool utilization. As this has a potential for major issues, we decided not to decrease assigned addresses immediately after receiving Decline, but to do it later when we recover the address back to the available pool.

8.11 Statistics in the DHCPv6 Server

> **Note**
> This section describes DHCPv6-specific statistics. For a general overview and usage of statistics, see Chapter 14.

The DHCPv6 server supports the following statistics:

8.12 Management API for the DHCPv6 Server

The management API allows the issuing of specific management commands, such as statistics retrieval, reconfiguration or shutdown. For more details, see Chapter 15. Currently the only supported communication channel type is UNIX stream socket. By default there are no sockets open. To instruct Kea to open a socket, the following entry in the configuration file can be used:

```
"Dhcp6": {
    "control-socket": {
        "socket-type": "unix",
        "socket-name": "/path/to/the/unix/socket"
    },

    "subnet6": [
        ...
    ],
    ...
}
```

The length of the path specified by the **socket-name** parameter is restricted by the maximum length for the unix socket name on your operating system, i.e. the size of the **sun_path** field in the **sockaddr_un** structure, decreased by 1. This value varies on different operating systems between 91 and 107 characters. Typical values are 107 on Linux and 103 on FreeBSD.

Communication over control channel is conducted using JSON structures. See the Control Channel section in the Kea Developer's Guide for more details.

The DHCPv6 server supports **statistic-get**, **statistic-reset**, **statistic-remove**, **statistic-get-all**, **statistic-reset-all** and **statistic-remove-all**, specified in Section 14.3. It also supports **list-commands** and **shutdown**, specified in Section 15.3.2 and Section 15.3.3, respectively.

8.13 Supported DHCPv6 Standards

The following standards are currently supported:

Statistic	Data Type	Description
pkt6-received	integer	Number of DHCPv6 packets received. This includes all packets: valid, bogus, corrupted, rejected etc. This statistic is expected to grow rapidly.
pkt6-receive-drop	integer	Number of incoming packets that were dropped. The exact reason for dropping packets is logged, but the most common reasons may be: an unacceptable or not supported packet type, direct responses are forbidden, the server-id sent by the client does not match the server's server-id or the packet is malformed.
pkt6-parse-failed	integer	Number of incoming packets that could not be parsed. A non-zero value of this statistic indicates that the server received a malformed or truncated packet. This may indicate problems in your network, faulty clients, faulty relay agents or a bug in the server.
pkt6-solicit-received	integer	Number of SOLICIT packets received. This statistic is expected to grow. Its increase means that clients that just booted started their configuration process and their initial packets reached your server.
pkt6-advertise-received	integer	Number of ADVERTISE packets received. Advertise packets are sent by the server and the server is never expected to receive them. A non-zero value of this statistic indicates an error occurring in the network. One likely cause would be a misbehaving relay agent that incorrectly forwards ADVERTISE messages towards the server rather back to the clients.
pkt6-request-received	integer	Number of REQUEST packets received. This statistic is expected to grow. Its increase means that clients that just booted received the server's response (ADVERTISE), accepted it and are now requesting an address (REQUEST).
pkt6-reply-received	integer	Number of REPLY packets received. This statistic is expected to remain zero at all times, as REPLY packets are sent by the server and the server is never expected to receive them. A non-zero value indicates an error. One likely cause would be a misbehaving relay agent that incorrectly forwards REPLY messages towards the server, rather back to the clients.
pkt6-renew-received	integer	Number of RENEW packets received. This statistic is expected to grow. Its increase means that clients received their addresses and prefixes and are trying to renew them.
pkt6-rebind-received	integer	Number of REBIND packets received. A non-zero value indicates that clients didn't receive responses to their RENEW messages (regular lease renewal mechanism) and are

- *Dynamic Host Configuration Protocol for IPv6*, RFC 3315: Supported messages are SOLICIT, ADVERTISE, REQUEST, RELEASE, RENEW, REBIND, INFORMATION-REQUEST, CONFIRM and REPLY.

- *IPv6 Prefix Options for Dynamic Host Configuration Protocol (DHCP) version 6*, RFC 3633: Supported options are IA_PD and IA_PREFIX. Also supported is the status code NoPrefixAvail.

- *DNS Configuration options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6)*, RFC 3646: Supported option is DNS_SERVERS.

- *The Dynamic Host Configuration Protocol for IPv6 (DHCPv6) Relay Agent Remote-ID Option*, RFC 4649: REMOTE-ID option is supported.

- *The Dynamic Host Configuration Protocol for IPv6 (DHCPv6) Client Fully Qualified Domain Name (FQDN) Option*, RFC 4704: Supported option is CLIENT_FQDN.

- *Relay-Supplied DHCP Options*, RFC 6422: Full functionality is supported: OPTION_RSOO, ability of the server to echo back the options, checks whether an option is RSOO-enabled, ability to mark additional options as RSOO-enabled.

- *Client Link-Layer Address Option in DHCPv6*, RFC 6939: Supported option is client link-layer address option.

- *Issues and Recommendations with Multiple Stateful DHCPv6 Options*, RFC 7550: All recommendations related to the DHCPv6 server operation are supported.

8.14 DHCPv6 Server Limitations

These are the current limitations of 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.

- 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.

- Client reconfiguration (RECONFIGURE) is not yet supported.

Chapter 9

Lease Expiration in DHCPv4 and DHCPv6

The primary role of the DHCP server is to assign addresses and/or delegate prefixes to DHCP clients. These addresses and prefixes are often referred to as "leases". Leases are typically assigned to clients for a finite amount of time, known as the "valid lifetime". DHCP clients who wish to continue using their assigned leases, will periodically renew them by sending the appropriate message to the DHCP server. The DHCP server records the time when these leases are renewed and calculates new expiration times for them.

If the client does not renew a lease before its valid lifetime elapses, the lease is considered expired. There are many situations when the client may cease lease renewals. A common scenario is when the machine running the client shuts down for an extended period of time.

The process through which the DHCP server makes expired leases available for reassignment is referred to as "lease reclamation" and expired leases returned to availability through this process are referred to as "reclaimed". The DHCP server attempts to reclaim an expired lease as soon as it detects that it has expired. One way in which the server detects expiration occurs when it is trying to allocate a lease to a client and finds this lease already present in the database but expired. Another way is by periodically querying the lease database for them. Regardless of how an expired lease is detected, before it may assigned to a client, it must be reclaimed.

This chapter explains how to configure the server to periodically query for the expired leases and how to minimize the impact of the periodic lease reclamation process on the server's responsiveness. Finally, it explains "lease affinity", which provides the means to assign the same lease to a returning client after its lease has expired.

Although, all configuration examples in this section are provided for the DHCPv4 server, the same parameters may be used for the DHCPv6 server configuration.

9.1 Lease Reclamation

Lease reclamation is the process through which an expired lease becomes available for assignment to the same or different client. This process involves the following steps for each reclaimed lease:

- Invoke callouts for the **lease4_expire** or **lease6_expire** hook points if hook libraries supporting those callouts are currently loaded.

- Update DNS, i.e. remove any DNS entries associated with the expired lease.

- Update lease information in the lease database to indicate that the lease is now available for re-assignment.

- Update counters on the server, which includes increasing the number of reclaimed leases and decreasing the number of assigned addresses or delegated prefixes.

Please refer to Chapter 10 to see how to configure DNS updates in Kea, and to Chapter 13 for information about using hooks libraries.

9.2 Configuring Lease Reclamation

Kea can be configured to periodically detect and reclaim expired leases. During this process the lease entries in the database are modified or removed. While this is happening the server will not process incoming DHCP messages to avoid issues with concurrent access to database information. As a result, the server will be unresponsive while lease reclamation is performed and DHCP queries will accumulate; responses will be sent once the leases reclamation cycle is complete.

In deployments where response time is critical, administrators may wish to minimize the interruptions in service caused by lease reclamation. Toward this end, Kea provides configuration parameters to control: the frequency of lease reclamation cycles, the maximum number of leases processed in a single reclamation cycle, and the maximum amount of time a single reclamation cycle is allowed to run before being interrupted. The following examples demonstrate how these parameters can be used:

```
"Dhcp4": {
    ...

    "expired-leases-processing": {
        "reclaim-timer-wait-time": 5,
        "max-reclaim-leases": 0,
        "max-reclaim-time": 0,
        "flush-reclaimed-timer-wait-time": 0,
    },

    ...
}
```

The first parameter is expressed in seconds and specifies an interval between the two consecutive lease reclamation cycles. This is explained by the following diagram.

```
|  c1  |              | c2 |              |c3|              | c4 |
|<---->|<---------->|<-->|<---------->|<>|<---------->|<-->|
------------------------------------------------------------>
|      |     5s     |    |     5s     |  |     5s     |    | time
```

This diagram shows four lease reclamation cycles (c1 through c4) of variable duration. Note that the duration of the reclamation cycle depends on the number of expired leases detected and processed in the particular cycle. This duration is also usually significantly shorter than the interval between the cycles.

According to the **reclaim-timer-wait-time** the server keeps fixed intervals of five seconds between the end of one cycle and the start of the next cycle. This guarantees the presence of 5s long periods during which the server remains responsive to DHCP queries and does not perform lease reclamation. The **max-reclaim-leases** and **max-reclaim-time** are set to 0, which sets no restriction on the maximum number of leases reclaimed in the particular cycle, or on the maximum duration of each cycle.

In deployments with high lease pool utilization, relatively short valid lifetimes, and frequently disconnecting clients which allow leases to expire, the number of expired leases requiring reclamation at any given time may rise significantly. In this case it is often desirable to apply restrictions on the maximum duration of a reclamation cycle or the maximum number of leases reclaimed in a cycle. The following configuration demonstrates how this can be done:

```
"Dhcp4": {
    ...

    "expired-leases-processing": {
        "reclaim-timer-wait-time": 3,
        "max-reclaim-leases": 100,
        "max-reclaim-time": 50,
        "unwarned-reclaim-cycles": 10,
    },

    ...
}
```

The **max-reclaim-leases** parameter limits the number of leases reclaimed in a single cycle to 100. The **max-reclaim-time** limits the maximum duration of each cycle to 50ms. The lease reclamation cycle will be interrupted if either of these limitations is reached. The reclamation of all unreclaimed leases will be attempted in subsequent cycles.

The following diagram illustrates the behavior of the system in the presence of many expired leases, when the limits are applied for the reclamation cycles.

```
| c1 |                    | c2 |                    | c3 |                    | c4 |
|<-->|<-------------->|<-->|<-------------->|<-->|<-------------->|<-->|<--
--------------------------------------------------------------------------->
|50ms|       3s       |50ms|       3s       |50ms|       3s       |50ms|  time
```

The diagram demonstrates the case when each reclamation cycle would take more than 50ms, and thus is interrupted according to the value of the **max-reclaim-time**. This results in equal durations of all reclamation cycles over time. Note that in this example the limitation of maximum 100 leases is not reached. This may be the case when database transactions are slow or callouts in the hook libraries attached to the server are slow. Regardless, the choosing values for either the maximum number of leases or a maximum cycle time strongly depends on the particular deployment, lease database backend being used, and any hooks libraries etc. Administrators may need to experiment to tune the system to suit the dynamics of their deployment.

It is important to realize that with the use of these limits, there is a risk that expired leases will accumulate faster than the server can reclaim them. This should not be the problem if the server is dealing with a temporary burst of expirations, because it should be able to eventually deal with them over time. However, if leases expire at a high rate for a longer period of time, the unreclaimed leases will pile up in the database. In order to notify the administrator that the current configuration does not satisfy the needs for reclamation of expired leases, the server issues a warning message in the log if it was unable to reclaim all leases within the last couple of reclamation cycles. The number of cycles after which such warning is issued is specified with the **unwarned-reclaim-cycles** configuration parameter.

Setting the **reclaim-timer-wait-time** to 0 disables periodic reclamation of the expired leases.

9.3 Configuring Lease Affinity

Suppose that a laptop goes to a sleep mode after a period of user inactivity. While the laptop is in sleep mode, its DHCP client will not renew leases obtained from the server and these leases will eventually expire. When the laptop wakes up, it is often desirable for it to continue using its previous assigned IP addresses. In order to facilitate this, the server needs to correlate returning clients with their expired leases When the client returns, the server will first check for those leases and re-assign them if they have not been assigned to another client. The ability of the server to re-assign the same lease to a returning client is referred to as "lease affinity".

When lease affinity is enabled, the server will still reclaim leases according to the parameters described in Section 9.2, but the reclaimed leases will be held in the database (rather than removed) for the specified amount of time. When the client returns, the server will first check if there are any reclaimed leases associated with this client and re-assign them if possible. However, it is important to note that any reclaimed lease may be assigned to another client if that client specifically asks for it. Therefore, the lease affinity does not guarantee that the reclaimed lease will be available for the client who used it before; it merely increases the chances for the client to be assigned the same lease. If the lease pool is small (this mostly applies to DHCPv4 for which address space is small), there is an increased likelihood that the expired lease will be assigned to another client.

Consider the following configuration:

```
"Dhcp4": {
    ...

    "expired-leases-processing": {
        "reclaim-timer-wait-time": 3,
        "hold-reclaimed-time": 1800,
        "flush-reclaimed-timer-wait-time": 5
    },

    ...
}
```

The **hold-reclaim-time** specifies how many seconds after an expiration a reclaimed lease should be held in the database for re-assignment to the same client. In the example given above, reclaimed leases will be held for 30 minutes (1800s) after their expiration. During this time, the server will likely be able to re-assign the same lease to the returning client, unless another client requests this lease and the server assigns it.

The server must periodically remove reclaimed leases for which the time indicated by **hold-reclaim-time** has elapsed. The **flush-reclaimed-timer-wait-time** controls how often the server removes such leases. In the example provided above, the server will initiate removal of such leases 5 seconds after the previous removal attempt was completed. Setting this value to 0 disables lease affinity, in which case leases will be removed from the lease database when they are reclaimed. If lease affinity is enabled, it is recommended that **hold-reclaim-time** be set to a value significantly higher than the **reclaim-timer-wait-time**, as timely removal of expired-reclaimed leases is less critical while the removal process may impact server responsiveness.

9.4 Default Configuration Values for Leases Reclamation

The following list presents all configuration parameters pertaining to processing expired leases with their default values:

- **reclaim-timer-wait-time** = 10 [seconds]

- **flush-reclaimed-timer-wait-time** = 25 [seconds]

- **hold-reclaimed-time** = 3600 [seconds]

- **max-reclaim-leases** = 100

- **max-reclaim-time** = 250 [milliseconds]

- **unwarned-reclaim-cycles** = 5

The default value for any parameter is used when this parameter not explicitly specified in the configuration. Also, the **expired-leases-processing** map may be omitted entirely in the configuration, in which case the default values are used for all parameters listed above.

9.5 Reclaiming Expired Leases with Command

The *leases-reclaim* command can be used to trigger leases reclamation at any time. Please consult the Section 15.3.1 for the details about using this command.

Chapter 10

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 the appropriate DNS servers, D2 must have a catalog of servers from which to select. In fact, D2 has two such catalogs, one for forward DNS and one for reverse DNS; these catalogs are referred to as DDNS Domain Lists. Each list consists of one or more named DDNS Domains. Further, each DDNS Domain has a list of one or more DNS servers that publish the DNS data for that domain.

When conducting forward domain matching, D2 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.

10.1   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, it is a separate binary that can be run on its own or through **keactrl** (see Chapter 6). In normal operation, controlling **kea-dhcp-ddns** with **keactrl** is recommended. However, it is also possible to run the DHCP-DDNS server directly. It accepts the following command-line switches:

- **-c** *file* - specifies the configuration file. This is the only mandatory switch.

- **-d** - specifies whether the server logging should be switched to debug/verbose mode. In verbose mode, the logging severity and debuglevel specified in the configuration file are ignored and "debug" severity and the maximum debuglevel (99) are assumed. The flag is convenient, for temporarily switching the server into maximum verbosity, e.g. when debugging.

- **-v** - prints out Kea version and exits.

- **-W** - prints out the Kea configuration report and exits. The report is a copy of the `config.report` file produced by `./configure`: it is embedded in the executable binary.

- **-W** - prints out Kea configuration report and exits.

The `config.report` may also be accessed more directly. The following command may be used to extract this information. The binary **path** may be found in the install directory or in the `.libs` subdirectory in the source tree. For example `kea/src/bin/d2/.libs/kea-dhcp-ddns`.

```
strings path/kea-dhcp-ddns | sed -n 's/;;;; //p'
```

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

During startup the server will attempt to create a PID file of the form: [localstatedir]/[conf name].kea-dhcp-ddns.pid where:

- **localstatedir**: The value as passed into the build configure script. It defaults to "/usr/local/var". Note that this value may be overridden at run time by setting the environment variable KEA_PIDFILE_DIR. This is intended primarily for testing purposes.

- **conf name**: The configuration file name used to start the server, minus all preceding path and file extension. For example, given a pathname of "/usr/local/etc/kea/myconf.txt", the portion used would be "myconf".

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

10.2   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

10.2.1 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** - Socket protocol to use when sending requests to D2. Currently only UDP is supported. TCP may be available in a future release.

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

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.

10.2.2 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.

- **digest-bits** - is used to specify the minimum truncated length in bits. The default value 0 means truncation is forbidden, non-zero values must be an integral number of octets, be greater than 80 and the half of the full length. Note in BIND9 this parameter is appended after a dash to the algorithm name.

- **secret** - is used to specify the shared secret key code for this key. This value is case sensitive and must exactly match the value specified on the DNS server(s). It is a base64-encoded text value.

As an example, suppose that a domain D2 will be updating is maintained by a 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.

10.2.3 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.

10.2.3.1 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.

This section describes how to add Forward DDNS Domains. Repeat these steps for each Forward DDNS Domain desired. Each Forward DDNS Domain has the following parameters:

- **name** - The fully qualified domain name (or zone) that this DDNS Domain can update. This 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.

10.2.3.1.1 Adding Forward DNS Servers

This 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 in 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.

10.2.4 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.

10.2.4.1 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.

This section describes how to add Reverse DDNS Domains. Repeat these steps for each Reverse DDNS Domain desired. Each Reverse DDNS Domain has the following parameters:

- **name** - The fully qualified reverse zone that this DDNS Domain can update. This is the value used during reverse matching which will compare it with a reversed version of the request's lease address. The zone name should follow the appropriate standards: for example, to to support the IPv4 subnet 172.16.1, the name should be. "1.16.172.in-addr.arpa.". Similarly, to support an IPv6 subnet of 2001:db8:1, the name should be "1.0.0.0.8.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.

10.2.4.1.1 Adding Reverse DNS Servers

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

```
"DhcpDdns": {
    "reverse-ddns": {
  "ddns-domains": [
      {
    "name": "1.0.0.0.8.B.D.0.1.0.0.2.ip6.arpa.",
    "key-name": "",
    "dns-servers": [
        {
      "hostname": "",
      "ip-address": "172.88.99.10",
      "port": 53
        }
    ]
      }
  ]
    }
}
```

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

10.2.5 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 10.1: Our example network

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

#	DDNS Domain Name	DNS Servers
1.	four.example.com.	172.16.1.5, 172.16.2.5
2.	six.example.com.	3001:1::50
3.	example.com.	172.16.2.5

Table 10.2: Forward DDNS Domains Needed

match. The FQDN, "myhost.four.example.com.", will match the first domain ("four.example.com") while "admin.example.com." will match the third domain ("example.com"). The FQDN, "other.example.net." will fail to match any domain and would be rejected.

The following example configuration specified the Forward DDNS Domains.

```
"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 10.3: 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>
```

10.3   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 11

The LFC process

11.1 Overview

kea-lfc is a service process that removes redundant information from the files used to provide persistent storage for the memfile data base backend. This service is written to run as a stand alone process.

While **kea-lfc** can be started externally, there is usually no need to do this. **kea-lfc** is run on a periodic basis by the Kea DHCP servers.

The process operates on a set of files, using them for input and output of the lease entries and to indicate where it is in the process in case of an interruption. Currently the caller must supply names for all of the files, in the future this requirement may be relaxed with the process getting the names from either the configuration file or from defaults.

11.2 Command Line Options

kea-lfc is run as follows:

```
kea-lfc [-4 | -6] -c config-file -p pid-file -x previous-file -i copy-file -o output-file - ←
    f finish-file
```

The argument **-4** or **-6** selects the protocol version of the lease files.

The **-c** argument specifies the configuration file. This is required, but not currently used by the process.

The **-p** argument specifies the PID file. When the **kea-lfc** process starts it attempts to determine if another instance of the process is already running by examining the pid file. If one is already running the new process is terminated. If one isn't running it writes its pid into the pid file.

The other filenames specify where the **kea-lfc** process should look for input, write its output and use for bookkeeping.

- **previous** — When **kea-lfc** starts this is the result of any previous run of **kea-lfc**. When **kea-lfc** finishes it is the result of this run. If **kea-lfc** is interrupted before completing, this file may not exist.

- **input** — Before the DHCP server invokes **kea-lfc** it will move the current lease file here and then call **kea-lfc** with this file.

- **output** — The temporary file **kea-lfc** should use to write the leases. Upon completion of writing this file, it will be moved to the finish file (see below).

- **finish** — Another temporary file **kea-lfc** uses for bookkeeping. When **kea-lfc** completes writing the outputfile it moves it to this file name. After **kea-lfc** finishes deleting the other files (previous and input) it moves this file to previous lease file. By moving the files in this fashion the **kea-lfc** and the DHCP server processes can determine the correct file to use even if one of the processes was interrupted before completing its task.

There are several additional arguments mostly for debugging purposes. **-d** Sets the logging level to debug. **-v** and **-V** print out version stamps with **-V** providing a longer form. **-h** prints out the usage string.

Chapter 12

Client Classification

12.1   Client Classification Overview

In certain cases it is useful to differentiate between different types of clients and treat them accordingly. Common reasons include:

- The clients represent different pieces of topology, e.g. a cable modem is different to the clients behind that modem.

- The clients have different behavior, e.g. a smart phone behaves differently to a laptop.

- The clients require different values for some options, e.g. a docsis3.0 cable modem requires different settings to docsis2.0 cable modem.

It is envisaged that client classification will be used for changing the behavior of almost any part of the DHCP message processing, including the assignment of leases from different pools, the assignment of different options (or different values of the same options) etc. In the current release of the software however, there are only three mechanisms that take advantage of client classification: subnet selection, assignment of different options and, for DHCPv4 cable modems, the setting of specific options for use with the TFTP server address and the boot file field.

The process of doing classification is conducted in three steps:

1. Assess an incoming packet and assign it to zero or more classes.

2. Choose a subnet, possibly based on the class information.

3. Assign options, again possibly based on the class information.

When determining which options to include in the response the server will examine the union of options from all of the assigned classes. In the case two or more classes include the same option, the value from the first class examined will be used. When choosing a subnet, the server will iterate over all of the subnets that are feasible given the information found in the packet (client address, relay address etc). It will use the first subnet it finds that either doesn't have a class associated with it or that has a class which matches one of the packet's classes. In the future the processing order of the various classes may be specified but for now it is being left unspecified and may change in future releases.

As an example, imagine that an incoming packet matches two classes. Class "foo" defines values for an NTP server (option 42 in DHCPv4) and an SMTP server (option 69 in DHCPv4) while class "bar" defines values for an NTP server and a POP3 server (option 70 in DHCPv4). The server will examine the three options NTP, SMTP and POP3 and return any of them that the client requested. As the NTP server was defined twice the server will choose only one of the values for the reply: the class from which the value is obtained is unspecified.

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

Note

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

12.2 Using Static Host Reservations In Classification

Classes can be statically assigned to the clients using techniques described in Section 7.3.6 and Section 8.3.5.

12.3 Using Vendor Class Information In Classification

The server checks whether an incoming DHCPv4 packet includes the vendor class identifier option (60) or an incoming DHCPv6 packet includes the vendor class option (16). If it does, the content of that option is prepended with "VENDOR_CLASS_" and the result is interpreted as a class. For example, modern cable modems will send this option with value "docsis3.0" and so the packet will belong to class "VENDOR_CLASS_docsis3.0".

12.4 Using Expressions In Classification

The expression portion of classification contains operators and values. All values are currently strings and operators take a string or strings and return another string. When all the operations have completed the result should be a value of "true" or "false". The packet belongs to the class (and the class name is added to the list of classes) if the result is "true". Expressions are written in standard format and can be nested.

Expressions are pre-processed during the parsing of the configuration file and converted to an internal representation. This allows certain types of errors to be caught and logged during parsing. Examples of these errors include an incorrect number or types of arguments to an operator. The evaluation code will also check for this class of error and generally throw an exception, though this should not occur in a normally functioning system.

Other issues, for example the starting position of a substring being outside of the substring or an option not existing in the packet, result in the operator returning an empty string.

Expressions are a work in progress and the supported operators and values are limited. The expectation is that additional operators and values will be added over time, however the basic mechanisms will remain the same.

Notes:

- Hexadecimal strings are converted into a string as expected. The starting "0X" or "0x" is removed and if the string is an odd number of characters a "0" is prepended to it.

- IP addresses are converted into strings of length 4 or 16. IPv4, IPv6, and IPv4 embedded IPv6 (e.g., IPv4 mapped IPv6) addresses are supported.

- Integers in an expression are converted to 32 bit unsigned integers and are represented as four-byte strings. For example 123 is represented as 0x0000007b. All expressions that return numeric values use 32-bit unsigned integers, even if the field in the packet is smaller. In general it is easier to use decimal notation to represent integers, but it is also possible to use hex notation. When using hex notation to represent an integer care should be taken to make sure the value is represented as 32 bits, e.g. use 0x00000001 instead of 0x1 or 0x01. Also, make sure the value is specified in network order, e.g. 1 is represented as 0x00000001.

- "option[code].hex" extracts the value of the option with the code "code" from the incoming packet. If the packet doesn't contain the option, it returns the empty string. The string is presented as a byte string of the option payload without the type code or length fields.

- "option[code].exists" checks if an option with the code "code" is present in the incoming packet. It can be used with empty options.

Name	Example expression	Example value
String literal	'example'	'example'
Hexadecimal string literal	0x5a7d	'Z}'
IP address literal	10.0.0.1	0x0a000001
Integer literal	123	'123'
Integer literal	123	'123'
Binary content of the option	option[123].hex	'(content of the option)'
Option existence	option[123].exists	'true'
DHCPv4 relay agent sub-option	relay4[123].hex	'(content of the RAI sub-option)'
DHCPv6 Relay Options	relay6[nest].option[code].hex	(value of the option)
DHCPv6 Relay Peer Address	relay6[nest].peeraddr	2001:DB8::1
DHCPv6 Relay Link Address	relay6[nest].linkaddr	2001:DB8::1
Interface name of packet	pkt.iface	eth0
Source address of packet	pkt.src	10.1.2.3
Destination address of packet	pkt.dst	10.1.2.3
Length of packet	pkt.len	513
Hardware address in DHCPv4 packet	pkt4.mac	0x010203040506
Hardware length in DHCPv4 packet	pkt4.hlen	6
Hardware type in DHCPv4 packet	pkt4.htype	6
ciaddr field in DHCPv4 packet	pkt4.ciaddr	192.0.2.1
giaddr field in DHCPv4 packet	pkt4.giaddr	192.0.2.1
yiaddr field in DHCPv4 packet	pkt4.yiaddr	192.0.2.1
siaddr field in DHCPv4 packet	pkt4.siaddr	192.0.2.1
Message Type in DHCPv4 packet	pkt4.msgtype	1
Transaction ID (xid) in DHCPv4 packet	pkt4.transid	12345
Message Type in DHCPv6 packet	pkt6.msgtype	1
Transaction ID in DHCPv6 packet	pkt6.transid	12345
Vendor option existence (any vendor)	vendor[*].exists	true
Vendor option existence (specific vendor)	vendor[4491].exists	true
Enterprise-id from vendor option	vendor.enterprise	4491
Vendor sub-option existence	vendor[4491].option[1].exists	true
Vendor sub-option content	vendor[4491].option[1].hex	docsis3.0
Vendor class option existence (any vendor)	vendor-class[*].exists	true
Vendor class option existence (specific vendor)	vendor-class[4491].exists	true
Enterprise-id from vendor class option	vendor-class.enterprise	4491
First data chunk from vendor class option	vendor-class[4491].data	docsis3.0
Specific data chunk from vendor class option	vendor-class[4491].data[3]	docsis3.0

Table 12.1: List of Classification Values

- "relay4[code].hex" attempts to extract the value of the sub-option "code" from the option inserted as the DHCPv4 Relay Agent Information (82) option. If the packet doesn't contain a RAI option, or the RAI option doesn't contain the requested sub-option, the expression returns an empty string. The string is presented as a byte string of the option payload without the type code or length fields. This expression is allowed in DHCPv4 only.

- "relay4" shares the same representation types as "option", for instance "relay4[code].exists" is supported.

- "relay6[nest]" allows access to the encapsulations used by any DHCPv6 relays that forwarded the packet. The "nest" level specifies the relay from which to extract the information, with a value of 0 indicating the relay closest to the DHCPv6 server. If the requested encapsulation doesn't exist an empty string "" is returned. This expression is allowed in DHCPv6 only.

- "relay6[nest].option[code]" shares the same representation types as "option", for instance "relay6[nest].option[code].exists" is supported.

- Expressions starting with "pkt4" can be used only in DHCPv4. They allows access to DHCPv4 message fields.

- "pkt6" refers to information from the client request. To access any information from an intermediate relay use "relay6". "pkt6.msgtype" and "pkt6.transid" output a 4 byte binary string for the message type or transaction id. For example the message type SOLICIT will be "0x00000001" or simply 1 as in "pkt6.msgtype == 1".

- Vendor option means Vendor-Identifying Vendor-specific Information option in DHCPv4 (code 125, see Section 4 of RFC 3925) and Vendor-specific Information Option in DHCPv6 (code 17, defined in Section 22.17 of RFC 3315). Vendor class option means Vendor-Identifying Vendor Class Option in DHCPv4 (code 124, see Section 3 of RFC 3925) in DHCPv4 and Class Option in DHCPv6 (code 16, see Section 22.16 of RFC 3315). Vendor options may have sub-options that are referenced by their codes. Vendor class options do not have sub-options, but rather data chunks, which are referenced by index value. Index 0 means the first data chunk, Index 1 is for the second data chunk (if present), etc.

- In the vendor and vendor-class constructs Asterisk (*) or 0 can be used to specify a wildcard enterprise-id value, i.e. it will match any enterprise-id value.

- Vendor Class Identifier (option 60 in DHCPv4) can be accessed using option[60] expression.

- RFC3925 and RFC3315 allow for multiple instances of vendor options to appear in a single message. The client classification code currently examines the first instance if more than one appear. For vendor.enterprise and vendor-class.enterprise expressions, the value from the first instance is returned. Please submit a feature request on Kea website if you need support for multiple instances.

Name	Example	Description
Equal	'foo' == 'bar'	Compare the two values and return "true" or "false"
Not	not ('foo' == 'bar')	Logical negation
And	('foo' == 'bar') and ('bar' == 'foo')	Logical and
Or	('foo' == 'bar') or ('bar' == 'foo')	Logical or
Substring	substring('foobar',0,3)	Return the requested substring
Concat	concat('foo','bar')	Return the concatenation of the strings

Table 12.2: List of Classification Expressions

12.4.1 Logical operators

The Not, And and Or logical operators are the common operators. Not has the highest precedence and Or the lowest. And and Or are (left) associative, parentheses around a logical expression can be used to enforce a specific grouping, for instance in "A and (B or C)" (without parentheses "A and B or C" means "(A and B) or C").

12.4.2 Substring

The substring operator "substring(value, start, length)" accepts both positive and negative values for the starting position and the length. For "start", a value of 0 is the first byte in the string while -1 is the last byte. If the starting point is outside of the original string an empty string is returned. "length" is the number of bytes to extract. A negative number means to count towards the beginning of the string but doesn't include the byte pointed to by "start". The special value "all" means to return all bytes from start to the end of the string. If length is longer than the remaining portion of the string then the entire remaining portion is returned. Some examples may be helpful:

```
substring('foobar', 0, 6) == 'foobar'
substring('foobar', 3, 3) == 'bar'
substring('foobar', 3, all) == 'bar'
substring('foobar', 1, 4) == 'ooba'
substring('foobar', -5, 4) == 'ooba'
substring('foobar', -1, -3) == 'oba'
substring('foobar', 4, -2) == 'ob'
substring('foobar', 10, 2) == ''
```

12.4.3 Concat

The concat function "concat(string1, string2)" returns the concatenation of its two arguments. For instance:

```
concat('foo', 'bar') == 'foobar'
```

Note

The expression for each class is executed on each packet received. If the expressions are overly complex, the time taken to execute them may impact the performance of the server. If you need complex or time consuming expressions you should write a hook to perform the necessary work.

12.5 Configuring Classes

A class contains three items: a name, a test expression and option data. The name must exist and must be unique amongst all classes. The test expression and option data are optional.

The test expression is a string containing the logical expression used to determine membership in the class. The entire expression is in double quotes.

The option data is a list which defines any options that should be assigned to members of this class.

In the following example the class named "Client_foo" is defined. It is comprised of all clients whose client ids (option 61) start with the string "foo". Members of this class will be given 192.0.2.1 and 192.0.2.2 as their domain name servers.

```
"Dhcp4": {
    "client-classes": [
        {
            "name": "Client_foo",
            "test": "substring(option[61].hex,0,3) == 'foo'",
            "option-data": [
                {
                    "name": "domain-name-servers",
                    "code": 6,
                    "space": "dhcp4",
                    "csv-format": true,
                    "data": "192.0.2.1, 192.0.2.2"
                }
            ]
```

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

This example shows a client class being defined for use by the DHCPv6 server. In it the class named "Client_enterprise" is defined. It is comprised of all clients who's client identifiers start with the given hex string (which would indicate a DUID based on an enterprise id of 0xAABBCCDD). Members of this class will be given an 2001:db8:0::1 and 2001:db8:2::1 as their domain name servers.

```
"Dhcp6": {
    "client-classes": [
        {
            "name": "Client_enterprise",
            "test": "substring(option[1].hex,0,6) == 0x0002AABBCCDD'",
            "option-data": [
                {
                    "name": "dns-servers",
                    "code": 23,
                    "space": "dhcp6",
                    "csv-format": true,
                    "data": "2001:db8:0::1, 2001:db8:2::1"
                }
            ]
        },
        ...
    ],
    ...
}
```

12.6 Configuring Subnets With Class Information

In certain cases it beneficial to restrict access to certain subnets only to clients that belong to a given class, using the "client-class" keyword when defining the subnet.

Let's assume that the server is connected to a network segment that uses the 192.0.2.0/24 prefix. The Administrator of that network has decided that addresses from range 192.0.2.10 to 192.0.2.20 are going to be managed by the DHCP4 server. Only clients belonging to client class Client_foo are allowed to use this subnet. Such a configuration can be achieved in the following way:

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

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

The following example shows restricting access to a DHCPv6 subnet. This configuration will restrict use of the addresses 2001:db8:1::1 to 2001:db8:1::FFFF to members of the "Client_enterprise" class.

```
"Dhcp6": {
    "client-classes": [
        {
            "name": "Client_enterprise",
            "test": "substring(option[1].hex,0,6) == 0x0002AABBCCDD'",
            "option-data": [
                {
                    "name": "dns-servers",
                    "code": 23,
                    "space": "dhcp6",
                    "csv-format": true,
                    "data": "2001:db8:0::1, 2001:db8:2::1"
                }
            ]
        },
        ...
    ],
    "subnet6": [
        {
            "subnet": "2001:db8:1::/64",
            "pools": [ { "pool": "2001:db8:1::-2001:db8:1::ffff" } ],
            "client-class": "Client_enterprise"
        }
    ],
    ...
}
```

12.7 Using Classes

Currently classes can be used for two functions. They can supply options to the members of the class and they can be used to choose a subnet from which an address will be assigned to the class member.

When supplying options, options defined as part of the class definition are considered "class globals". They will override any global options that may be defined and in turn will be overridden by any options defined for an individual subnet.

12.8 Classes and Hooks

You may use a hook to classify your packets. This may be useful if the expression would either be complex or time consuming and be easier or better to write as code. Once the hook has added the proper class name to the packet the rest of the classification system will work as normal in choosing a subnet and selecting options. For a description of hooks see Chapter 13, for a description on configuring classes see Section 12.5 and Section 12.6.

12.9 Debugging Expressions

While you are constructing your classification expressions you may find it useful to enable logging see Chapter 17 for a more complete description of the logging facility.

To enable the debug statements in the classifciaton system you will need to set the severity to "DEBUG" and the debug level to at least 55. The specific loggers are "kea-dhcp4.eval" and "kea-dhcp6.eval".

In order to understand the logging statements one must understand a bit about how expressions are evaluated, for a more complete description refer to the design document at http://kea.isc.org/wiki/KeaDesigns. In brief there are two structures used during the evaluation of an expression: a list of tokens which represent the expressions and a value stack which represents the values being manipulated.

The list of tokens is created when the configuration file is processed with most expressions and values being converted to a token. The list is organized in reverse Polish notation. During execution, the list will be traversed in order. As each token is executed it will be able to pop values from the top of the stack and eventually push its result on the top of the stack. Imagine the following expression:

```
"test": "substring(option[61].hex,0,3) == 'foo'",
```

This will result in the following tokens:

```
option, number (0), number (3), substring, text ('foo'), equals
```

In this example the first three tokens will simply push values onto the stack. The substring token will then remove those three values and compute a result that it places on the stack. The text option also places a value on the stack and finally the equals token removes the two tokens on the stack and places its result on the stack.

When debug logging is enabled, each time a token is evaluated it will emit a log message indicating the values of any objects that were popped off of the value stack and any objects that were pushed onto the value stack.

The values will be displayed as either text if the command is known to use text values or hexadecimal if the command either uses binary values or can manipulate either text or binary values. For expressions that pop multiple values off the stack, the values will be displayed in the order they were popped. For most expressions this won't matter but for the concat expression the values are displayed in reverse order from how they are written in the expression.

Let us assume that the following test has been entered into the configuration. This example skips most of the configuration to concentrate on the test.

```
"test": "substring(option[61].hex,0,3) == 'foo'",
```

The logging might then resemble this:

```
2016-05-19 13:35:04.163 DEBUG [kea.eval/44478] EVAL_DEBUG_OPTION Pushing option 61  ↩
    with value 0x666F6F626172
2016-05-19 13:35:04.164 DEBUG [kea.eval/44478] EVAL_DEBUG_STRING Pushing text string ↩
    '0'
2016-05-19 13:35:04.165 DEBUG [kea.eval/44478] EVAL_DEBUG_STRING Pushing text string ↩
    '3'
2016-05-19 13:35:04.166 DEBUG [kea.eval/44478] EVAL_DEBUG_SUBSTRING Popping length  ↩
    3, start 0, string 0x666F6F626172 pushing result 0x666F6F
2016-05-19 13:35:04.167 DEBUG [kea.eval/44478] EVAL_DEBUG_STRING Pushing text string ↩
    'foo'
2016-05-19 13:35:04.168 DEBUG [kea.eval/44478] EVAL_DEBUG_EQUAL Popping 0x666F6F and ↩
    0x666F6F pushing result 'true'
```

Note

The debug logging may be quite verbose if you have a number of expressions to evaluate. It is intended as an aid in helping you create and debug your expressions. You should plan to disable debug logging when you have your expressions working correctly. You also may wish to include only one set of expressions at a time in the configuration file while debugging them in order to limit the log statements. For example when adding a new set of expressions you might find it more convenient to create a configuration file that only includes the new expressions until you have them working correctly and then add the new set to the main configuration file.

Chapter 13

Hooks Libraries

13.1 Introduction

Although Kea offers a lot of flexibility, there may be cases where its behavior needs customisation. To accommodate this possibility, Kea includes the idea of "Hooks". This feature lets Kea load one or more dynamically-linked libraries (known as "hooks libraries") and, at various points in its processing ("hook points"), call functions in them. Those functions perform whatever custom processing is required.

Hooks libraries are attached to individual Kea processes, not to Kea as a whole. This means (for example) that it is possible to associate one set of libraries with the DHCP4 server and a different set to the DHCP6 server.

Another point to note is that it is possible for a process to load multiple libraries. When processing reaches a hook point, Kea calls the hooks library functions attached to it. If multiple libraries have attached a function to a given hook point, Kea calls all of them, in the order in which the libraries are specified in the configuration file. The order may be important: consult the documentation of the libraries to see if this is the case.

The next section describes how to configure hooks libraries. If you are interested in writing your own hooks library, information can be found in the Kea Developer's Guide.

13.2 Configuring Hooks Libraries

The hooks libraries for a given process are configured using the **hooks-libraries** keyword in the configuration for that process. (Note that the word "hooks" is plural). The value of the keyword is an array of map structures, each structure corresponding to a hooks library. For example, to set up two hooks libraries for the DHCPv4 server, the configuration would be:

```
"Dhcp4": {
    :
    "hooks-libraries": [
        {
            "library": "/opt/charging.so"
        },
        {
            "library": "/opt/local/notification.so",
            "parameters": {
                "mail": "spam@example.com",
                "floor": 13,
                "debug": false,
                "users": [ "alice", "bob", "charlie" ],
                "languages": {
                    "french": "bonjour",
                    "klingon": "yl'el"
                }
```

```
                }
            }
        ]
        :
}
```

Note

This is a change to the syntax used in Kea 0.9.2 and earlier, where hooks-libraries was a list of strings, each string being the name of a library. The change was made in Kea 1.0 to facilitate the specification of library-specific parameters, a capability available in Kea 1.1.0 onwards.

Note

The library reloading behavior has changed in Kea 1.1. Libraries are reloaded, even if their list hasn't changed. Kea does that, because the parameters specified for the library (or the files those parameters point to) may have changed.

Libraries may have additional parameters. Those are not mandatory in the sense that there may be libraries that don't require them. However, for specific library there is often specific requirement for specify certain set of parameters. Please consult the documentation for your library for details. In the example above, the first library has no parameters. The second library has five parameters, specifying mail (string parameter), floor (integer parameter), debug (boolean parameter) and even lists (list of strings) and maps (containing strings). Nested parameters could be used if the library supports it. This topic is explained in detail in the Hooks Developer's Guide in the "Configuring Hooks Libraries" section.

Notes:

- The full path to each library should be given.

- As noted above, order may be important - consult the documentation for each library.

- An empty list has the same effect as omitting the **hooks-libraries** configuration element all together.

  Note

  There is one case where this is not true: if Kea is running with a configuration that contains a **hooks-libraries** item, and that item is removed and the configuration reloaded, the removal will be ignored and the libraries remain loaded. As a workaround, instead of removing the **hooks-libraries** item, change it to an empty list. This will be fixed in a future version of Kea.

At the present time, only the kea-dhcp4 and kea-dhcp6 processes support hooks libraries.

13.3 Available Hooks Libraries

As described above, the hooks functionality provides a way to customize a Kea server without modifying the core code. ISC has chosen to take advantage of this feature to provide functions that may only be useful to a subset of Kea users. To this end ISC has created some hooks libraries; these discussed in the following sections.

Note

Some of these libraries will be available with the base code while others will be shared with organizations supporting development of Kea , possibly as a 'benefit' or 'thank you' for helping to sustain the larger Kea project. If you would like to get access to those libraries, please consider taking out a support contract: this includes professional support, advance security notifications, input into our roadmap planning, and many other benefits, while helping making Kea sustainable in the long term.

Currently the following libraries are available or planned from ISC:

ISC hopes to see more hooks libraries become available as time progresses, both developed internally and externally. Since this list may evolve dynamically, we decided to keep it on a wiki page, available at this link: http://kea.isc.org/wiki/Hooks. If you are a developer or are aware of any hooks libraries not listed there, please send a note to the kea-users or kea-dev mailing lists and someone will update it.

Name	Availability	Since
user_chk	Kea sources	Kea 0.8
Forensic Logging	Support customers	Kea 1.1.0
Lightweight 4over6	Support customers	Autumn 2016

Table 13.1: List of available hooks libraries

13.3.1 user_chk: Checking User Access

The user_chk library is the first hooks library published by ISC. It attempts to serve several purposes:

- To assign "new" or "unregistered" users to a restricted subnet, while "known" or "registered" users are assigned to unrestricted subnets.

- To allow DHCP response options or vendor option values to be customized based upon user identity.

- To provide a real time record of the user registration activity which can be sampled by an external consumer.

- To serve as a demonstration of various capabilities possible using the hooks interface.

Once loaded, the library allows segregating incomings requests into known and unknown clients. For known clients, the packets are processed mostly as usual, except it is possible to override certain options being sent. That can be done on a per host basis. Clients that are not on the known hosts list will be treated as unknown and will be assigned to the last subnet defined in the configuration file.

As an example of use, this behavior may be used to put unknown users into a separate subnet that leads to a walled garden, where they can only access a registration portal. Once they fill in necessary data, their details are added to the known clients file and they get a proper address after their device is restarted.

Note

This library was developed several years before the host reservation mechanism has become available. Currently host reservation is much more powerful and flexible, but nevertheless the user_chk capability to consult and external source of information about clients and alter Kea's behavior is useful and remains of educational value.

The library reads the /tmp/user_chk_registry.txt file while being loaded and each time an incoming packet is processed. The file is expected to have each line contain a self-contained JSON snippet which must have the following two entries:

- **type**, whose value is "HW_ADDR" for IPv4 users or "DUID" for IPv6 users

- **id**, whose value is either the hardware address or the DUID from the equest formatted as a string of hex digits, with or without ":" delimiters.

and may have the zero or more of the following entries:

- **bootfile** whose value is the pathname of the desired file

- **tftp_server** whose value is the hostname or IP address of the desired server

A sample user registry file is shown below:

```
{ "type" : "HW_ADDR", "id" : "0c:0e:0a:01:ff:04", "bootfile" : "/tmp/v4bootfile" }
{ "type" : "HW_ADDR", "id" : "0c:0e:0a:01:ff:06", "tftp_server" : "tftp.v4.example.com" }
{ "type" : "DUID", "id" : "00:01:00:01:19:ef:e6:3b:00:0c:01:02:03:04", "bootfile" : "/tmp/ ←
    v6bootfile" }
{ "type" : "DUID", "id" : "00:01:00:01:19:ef:e6:3b:00:0c:01:02:03:06", "tftp_server" : " ←
    tftp.v6.example.com" }
```

As with any other hooks libraries provided by ISC, internals of the user_chk code are well documented. You can take a look at the Kea Developer's Guide section dedicated to the user_chk library that discusses how the code works internally. That, together with our general entries in Hooks Framework section should give you some pointers how to extend this library and perhaps even write your own from scratch.

13.3.2 Forensic Logging Hooks

This section describes the forensic log hooks library. This library povides hooks that record a detailed log of lease assignments and renewals into a set of log files. Currently this library is only available to ISC customers with a support contract.

In many legal jurisdictions companies, especially ISPs, must record information about the addresses they have leased to DHCP clients. This library is designed to help with that requirement. If the information that it records is sufficient it may be used directly. If your jurisdiction requires that you save a different set of information you may use it as a template or example and create your own custom logging hooks.

This logging is done as a set of hooks to allow it to be customized to any particular need. Modifying a hooks library is easier and safer than updating the core code. In addition by using the hooks features those users who don't need to log this information can leave it out and avoid any performance penalties.

13.3.2.1 Log File Naming

The names for the log files have the following form:

```
path/base-name.CCYYMMDD.txt
```

The "path" and "base-name" are supplied in the configuration as described below see Section 13.3.2.4. The next part of the name is the date the log file was started, with four digits for year, two digits for month and two digits for day. The file is rotated on a daily basis.

> **Note**
> When running Kea servers for both DHCPv4 and DHCPv6 the log names must be distinct. See the examples in Section 13.3.2.4.

13.3.2.2 DHCPv4 Log Entries

For DHCPv4 the library creates entries based on DHCPREQUEST messages and corresponding DHCPv4 leases intercepted by lease4_select (for new leases) and lease4_renew (for renewed leases) hooks.

An entry is a single string with no embedded end-of-line markers and has the following sections:

```
address duration device-id {client-info} {relay-info}
```

Where:

- address - the leased IPv4 address given out and whether it was assigned or renewed.

- duration - the lease lifetime expressed in days (if present), hours, minutes and seconds. A lease lifetime of 0xFFFFFFFF will be denoted with the text "infinite duration".

- device-id - the client's hardware address shown as numerical type and hex digit string.

- client-info - the DHCP client id option (61) if present, shown as a hex string.

- relay-info - for relayed packets the giaddr and the RAI circuit id and remote id options (option 82 sub options 1 and 2) if present. The circuit id and remote id are presented as hex strings

For instance (line breaks added for readability, they would not be present in the log file).

```
Address: 192.2.1.100 has been renewed for 1 hrs 52 min 15 secs to a device with
hardware address: hwtype=1 08:00:2b:02:3f:4e, client-id: 17:34:e2:ff:09:92:54
connected via relay at address: 192.2.16.33, identified by circuit-id:
68:6f:77:64:79 and remote-id: 87:f6:79:77:ef
```

13.3.2.3 DHCPv6 Log Entries

For DHCPv6 the library creates entries based on lease management actions intercepted by the lease6_select (for new leases), lease6_renew (for renewed leases) and lease6_rebind (for rebound leases).

An entry is a single string with no embedded end-of-line markers and has the following sections:

```
address duration device-id {relay-info}*
```

Where:

- address - the leased IPv6 address or prefix given out and whether it was assigned or renewed.

- duration - the lease lifetime expressed in days (if present), hours, minutes and seconds. A lease lifetime of 0xFFFFFFFF will be denoted with the text "infinite duration".

- device-id - the client's DUID and hardware address (if present).

- relay-info - for relayed packets the content of relay agent messages, remote id and subscriber id options (x and xx) if present.

For instance (line breaks added for readability, they would not be present in the log file).

```
Address:2001:db8:1:: has been assigned for 0 hrs 11 mins 53 secs to a device with
DUID: 17:34:e2:ff:09:92:54 and hardware address: hwtype=1 08:00:2b:02:3f:4e
(from Raw Socket) connected via relay at address: fe80::abcd for client on
link address: 3001::1, hop count: 1, identified by remote-id:
01:02:03:04:0a:0b:0c:0d:0e:0f and subscriber-id: 1a:2b:3c:4d:5e:6f
```

13.3.2.4 Configuring the Forensic Log Hooks

To use this functionality the hook library must be included in the configuration of the desired DHCP server modules. The legal_log library is installed alongside the Kea libraries in `[kea-install-dir]/lib` where `kea-install-dir` is determined by the "--prefix" option of the configure script. It defaults to `/usr/local`. Assuming the default value then, configuring kea-dhcp4 to load the legal_log library could be done with the following Kea4 configuration:

```
"Dhcp4": {
    "hooks-libraries": [
        {
            "library": "/usr/local/lib/libdhcp_legal_log.so",
            "parameters": {
                "path": "/var/kea/var",
                "base-name": "kea-forensic4"
            }
        },
        ...
    ]
}
```

To configure it for kea-dhcp6, the commands are simply as shown below:

```
"Dhcp6": {
    "hooks-libraries": [
        {
            "library": "/usr/local/lib/libdhcp_legal_log.so",
            "parameters": {
                "path": "/var/kea/var",
                "base-name": "kea-forensic6"
            }
        },
        ...
    ]
}
```

Two Hook Library parameters are supported:

- path - the directory in which the forensic file(s) will be written. The default value is `[prefix]/kea/var`. The directory must exist.

- base-name - an arbitrary value which is used in conjunction with the current system date to form the current foresnic file name. It defaults to `kea-legal`.

Chapter 14

Statistics

14.1 Statistics Overview

Both Kea DHCP servers support statistics gathering. A working DHCP server encounters various events that can cause certain statistics to be collected. For example, a DHCPv4 server may receive a packet (pkt4-received statistic increases by one) that after parsing was identified as a DHCPDISCOVER (pkt4-discover-received). The Server processed it and decided to send a DHCPOFFER representing its answer (pkt4-offer-sent and pkt4-sent statistics increase by one). Such events happen frequently, so it is not uncommon for the statistics to have values in high thousands. They can serve as an easy and powerful tool for observing a server's and network's health. For example, if pkt4-received statistic stops growing, it means that the clients' packets are not reaching the server.

There are four types of statistics:

- *integer* - this is the most common type. It is implemented as 64 bit integer (int64_t in C++), so it can hold any value between -2ˆ63 to 2ˆ63 -1.

- *floating point* - this type is intended to store floating point precision. It is implemented as double C++ type.

- *duration* - this type is intended for recording time periods. It uses boost::posix_time::time_duration type, which stores hours, minutes, seconds and microseconds.

- *string* - this type is intended for recording statistics in textual form. It uses std::string C++ type.

During normal operation, DHCPv4 and DHCPv6 servers gather statistics. For a list of DHCPv4 and DHCPv6 statistics, see Section 7.7 and Section 8.11, respectively.

To extract data from the statistics module, the control channel can be used. See Chapter 15 for details. It is possible to retrieve a single or all statistics, reset statistics (i.e. set to neutral value, typically zero) or even remove completely a single or all statistics. See section Section 14.3 for a list of statistic oriented commands.

14.2 Statistics Lifecycle

It is useful to understand how the Statistics Manager module works. When the server starts operation, the manager is empty and does not have any statistics. When **statistic-get-all** is executed, an empty list is returned. Once the server performs an operation that causes a statistic to change, the related statistic will be created. In the general case, once a statistic is recorded even once, it is kept in the manager, until explicitly removed, by **statistic-remove** or **statistic-remove-all** being called or the server is shut down. Per subnet statistics are explicitly removed when reconfiguration takes place.

Statistics are considered run-time properties, so they are not retained after server restart.

Removing a statistic that is updated frequently makes little sense as it will be re-added when the server code next records that statistic. The **statistic-remove** and **statistic-remove-all** commands are intended to remove statistics that are not expected to be

observed in the near future. For example, a misconfigured device in a network may cause clients to report duplicate addresses, so the server will report increasing values of pkt4-decline-received. Once the problem is found and the device is removed, the system administrator may want to remove the pkt4-decline-received statistic, so it won't be reported anymore. If a duplicate address is detected ever again, the server will add this statistic back.

14.3 Commands for Manipulating Statistics

There are several commands defined that can be used for accessing (-get), resetting to zero or neutral value (-reset) or even removing a statistic completely (-remove). The difference between reset and remove is somewhat subtle. The reset command sets the value of the statistic to zero or neutral value. After this operation, the statistic will have a value of 0 (integer), 0.0 (float), 0h0m0s0us (duration) or "" (string). When asked for, a statistic with the values mentioned will be returned. **Remove** removes a statistic completely, so the statistic will not be reported anymore. Please note that the server code may add it back if there's a reason to record it.

Note

The following sections describe commands that can be sent to the server: the examples are not fragments of a configuration file. For more information on sending commands to Kea, see Chapter 15.

14.3.1 statistic-get command

statistic-get command retrieves a single statistic. It takes a single string parameter called **name** that specifies the statistic name. An example command may look like this:

```
{
    "command": "statistic-get",
    "arguments": {
        "name": "pkt4-received"
    }
}
```

The server will respond with details of the requested statistic, with result set to 0 indicating success and the specified statistic as the value of "arguments" parameter. If the requested statistic is not found, the response will contain an empty map, i.e. only { } as argument, but the status code will still be set to success (0).

14.3.2 statistic-reset command

statistic-reset command sets the specified statistic to its neutral value: 0 for integer, 0.0 for float, 0h0m0s0us for time duration and "" for string type. It takes a single string parameter called **name** that specifies the statistic name. An example command may look like this:

```
{
    "command": "statistic-reset",
    "arguments": {
        "name": "pkt4-received"
    }
}
```

If the specific statistic is found and reset was successful, the server will respond with a status of 0, indicating success and an empty parameters field. If an error is encountered (e.g. requested statistic was not found), the server will return a status code of 1 (error) and the text field will contain the error description.

14.3.3 statistic-remove command

statistic-remove command attempts to delete a single statistic. It takes a single string parameter called **name** that specifies the statistic name. An example command may look like this:

```
{
    "command": "statistic-remove",
    "arguments": {
        "name": "pkt4-received"
    }
}
```

If the specific statistic is found and its removal was successful, the server will respond with a status of 0, indicating success and an empty parameters field. If an error is encountered (e.g. requested statistic was not found), the server will return a status code of 1 (error) and the text field will contain the error description.

14.3.4 statistic-get-all command

statistic-get-all command retrieves all statistics recorded. An example command may look like this:

```
{
    "command": "statistic-get-all",
    "arguments": { }
}
```

The server will respond with details of all recorded statistics, with result set to 0 indicating that it iterated over all statistics (even when the total number of statistics is zero).

14.3.5 statistic-reset-all command

statistic-reset command sets all statistics to their neutral values: 0 for integer, 0.0 for float, 0h0m0s0us for time duration and "" for string type. An example command may look like this:

```
{
    "command": "statistic-reset-all",
    "arguments": { }
}
```

If the operation is successful, the server will respond with a status of 0, indicating success and an empty parameters field. If an error is encountered, the server will return a status code of 1 (error) and the text field will contain the error description.

14.3.6 statistic-remove-all command

statistic-remove-all command attempts to delete all statistics. An example command may look like this:

```
{
    "command": "statistic-remove-all",
    "arguments": { }
}
```

If the removal of all statistics was successful, the server will respond with a status of 0, indicating success and an empty parameters field. If an error is encountered, the server will return a status code of 1 (error) and the text field will contain the error description.

Chapter 15

Management API

A classic approach to daemon configuration assumes that the server's configuration is stored in configuration files and, when the configuration is changed, the daemon is restarted. This approach has the significant disadvantage of introducing periods of downtime, when client traffic is not handled. Another risk is that if the new configuration is invalid for whatever reason, the server may refuse to start, which will further extend the downtime period until the issue is resolved.

To avoid such problems, both the DHCPv4 and DHCPv6 servers include support for a mechanism that allows on-line reconfiguration without requiring server shutdown. Both servers can be instructed to open control sockets, which is a communication channel. The server is able to receive commands on that channel, act on them and report back status. While the set of commands in Kea 1.1.0 is limited, the number is expected to grow over time.

Currently the only supported type of control channel is UNIX stream socket. For details how to configure it, see Section 7.8 and Section 8.12. It is likely that support for other control channel types will be added in the future.

15.1 Data Syntax

Communication over the control channel is conducted using JSON structures. If configured, Kea will open a socket and listen for incoming connections. A process connecting to this socket is expected to send JSON commands structured as follows:

```
{
    "command": "foo",
    "arguments": {
        "param1": "value1",
        "param2": "value2",
        ...
    }
}
```

command is the name of command to execute and is mandatory. **arguments** is a map of parameters required to carry out the given command. The exact content and format of the map is command specific.

The server will process the incoming command and then send a response of the form:

```
{
    "result": 0|1,
    "text": "textual description",
    "arguments": {
        "argument1": "value1",
        "argument2": "value2",
        ...
    }
}
```

result indicates the outcome of the command. A value of 0 means success while any non-zero value designates an error. Currently 1 is used as a generic error, but additional error codes may be added in the future. The **text** field typically appears when result is non-zero and contains a description of the error encountered, but it may also appear for successful results (that is command specific). **arguments** is a map of additional data values returned by the server which is specific to the command issued. The map is always present, even if it contains no data values.

15.2 Using the Control Channel

Kea does not currently provide a client for using the control channel. The primary reason for this is the expectation is that the entity using the control channel is typically an IPAM or similar network management/monitoring software which may have quite varied expectations regarding the client and is even likely to be written in languages different than C or C++. Therefore only examples are provided to show how one can take advantage of the API.

The easiest way is to use a tool called **socat**, a tool available from socat homepage, but it is also widely available in Linux and BSD distributions. Once Kea is started, one could connect to the control interface using the following command:

```
$ socat UNIX:/path/to/the/kea/socket -
```

where **/path/to/the/kea/socket** is the path specified in the **Dhcp4/control-socket/socket-name** parameter in the Kea configuration file. Text passed to **socat** will be sent to Kea and the responses received from Kea printed to standard output.

It is also easy to open UNIX socket programmatically. An example of such a simplistic client written in C is available in the Kea Developer's Guide, chapter Control Channel Overview, section Using Control Channel.

15.3 Commands Supported by Both the DHCPv4 and DHCPv6 Servers

15.3.1 leases-reclaim

leases-reclaim command instructs the server to reclaim all expired leases immediately. The command has the following JSON syntax:

```
{
    "command": "leases-reclaim",
    "arguments": {
        "remove": true
    }
}
```

The *remove* boolean parameter is mandatory and it indicates whether the reclaimed leases should be removed from the lease database (if true), or they should be left in the *expired-reclaimed* state (if false). The latter facilitates lease affinity, i.e. ability to re-assign expired lease to the same client which used this lease before. See Section 9.3 for the details. Also, see Section 9.1 for the general information about the processing of expired leases (leases reclamation).

15.3.2 list-commands

The *list-commands* command retrieves a list of all commands supported by the server. It does not take any arguments. An example command may look like this:

```
{
    "command": "list-commands",
    "arguments": { }
}
```

The server will respond with a list of all supported commands. The arguments element will be a list of strings. Each string will convey one supported command.

15.3.3 shutdown

The *shutdown* command instructs the server to initiate its shutdown procedure. It is the equivalent of sending a SIGTERM signal to the process. This command does not take any arguments. An example command may look like this:

```
{
    "command": "shutdown",
    "arguments": { }
}
```

The server will respond with a confirmation that the shutdown procedure has been initiated.

Chapter 16

The libdhcp++ Library

libdhcp++ is a 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.

16.1   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. 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 17

Logging

17.1 Logging Configuration

During its operation Kea may produce many messages. They differ in severity (some are more important than others) and source (some are produced by specific components, e.g. hooks). It is useful to understand which log messages are needed and which are not, and configure your logging appropriately. For example, debug level messages can be safely ignored in a typical deployment. They are, however, very useful when debugging a problem.

The logging system in Kea is configured through the Logging section in your configuration file. All daemons (e.g. DHCPv4 and DHCPv6 servers) will use the configuration in the Logging section to see what should be logged and to where. This allows for sharing identical logging configuration between daemons.

17.1.1 Loggers

Within Kea, a message is logged through an entity called a "logger". Different components log messages through different loggers, and each logger can be configured independently of one another. Some components, in particular the DHCP server processes, may use multiple loggers to log messages pertaining to different logical functions of the component. For example, the DHCPv4 server uses one logger for messages pertaining to packet reception and transmission, another logger for messages related to lease allocation and so on. Some of the libraries used by the Kea servers, e.g. libdhcpsrv, use their own loggers.

Users implementing hooks libraries (code attached to the server at runtime) are responsible for creating the loggers used by those libraries. Such loggers should have unique names, different from the logger names used by Kea. In this way the messages output by the hooks library can be distinguished from messages issued by the core Kea code. Unique names also allow the loggers to be configured independently of loggers used by Kea. Whenever it makes sense, a hook library can use multiple loggers to log messages pertaining to different logical parts of the library.

In the Logging section of a configuration file you can specify the configuration for zero or more loggers (including loggers used by the proprietary hooks libraries). If there are no loggers specified, the code will use default values: these cause Kea to log messages of INFO severity or greater to standard output. There is also a small time window after Kea has been started, but has not yet read its configuration. Logging in this short period can be controlled using environment variables. For details, see Section 17.1.3.

The three main elements of a logger configuration are: **name** (the component that is generating the messages), the **severity** (what to log), and the **output_commands** (where to log). There is also a **debuglevel** element, which is only relevant if debug-level logging has been selected.

17.1.1.1 name (string)

Each logger in the system has a name, the name being that of the component binary file using it to log messages. For instance, if you want to configure logging for the DHCPv4 server, you add an entry for a logger named "kea-dhcp4". This configuration

will then be used by the loggers in the DHCPv4 server, and all the libraries used by it (unless a library defines its own logger and there is specific logger configuration that applies to that logger).

When tracking down an issue with the server's operation, use of DEBUG logging is required to obtain the verbose output needed for problems diagnosis. However, the high verbosity is likely to overwhelm the logging system in cases when the server is processing high volume traffic. To mitigate this problem, use can be made of the fact that Kea uses multiple loggers for different functional parts of the server and that each of these can be configured independently. If the user is reasonably confident that a problem originates in a specific function of the server, or that the problem is related to the specific type of operation, they may enable high verbosity only for the relevant logger, so limiting the debug messages to the required minimum.

The loggers are associated with a particular library or binary of Kea. However, each library or binary may (and usually does) include multiple loggers. For example, the DHCPv4 server binary contains separate loggers for: packet parsing, for dropped packets, for callouts etc.

The loggers form a hierarchy. For each program in Kea, there is a "root" logger, named after the program (e.g. the root logger for kea-dhcp (the DHCPv4 server) is named kea-dhcp4. All other loggers are children of this logger, and are named accordingly, e.g. the the allocation engine in the DHCPv4 server logs messages using a logger called kea-dhcp4.alloc-engine.

This relationship is important as each child logger derives its default configuration from its parent root logger. In the typical case, the root logger configuration is the only logging configuration specified in the configuration file and so applies to all loggers. If an entry is made for a given logger, any attributes specified override those of the root logger, whereas any not specified are inherited from it.

To illustrate this, suppose you are using the DHCPv4 server with the root logger "kea-dhcp4" logging at the INFO level. In order to enable DEBUG verbosity for the DHCPv4 packet drops, you must create configuration entry for the logger called "kea-dhcp4.bad-packets" and specify severity DEBUG for this logger. All other configuration parameters may be omited for this logger if the logger should use the default values specified in the root logger's configuration.

If there are multiple logger specifications in the configuration that might match a particular logger, the specification with the more specific logger name takes precedence. For example, if there are entries for both "kea-dhcp4" and "kea-dhcp4.dhcpsrv", the DHCPv4 server — and all libraries it uses that are not dhcpsrv — will log messages according to the configuration in the first entry ("kea-dhcp4").

Currently defined loggers are:

- **kea-dhcp4** - the root logger for the DHCPv4 server. All components used by the DHCPv4 server inherit the settings from this logger.

- **kea-dhcp4.alloc-engine** - used by the lease allocation engine, which is responsible for managing leases in the lease database, i.e. create, modify and remove DHCPv4 leases as a result of processing messages from the clients.

- **kea-dhcp4.bad-packets** - used by the DHCPv4 server daemon for logging inbound client packets that were dropped or to which the server responded with a DHCPNAK. It allows administrators to configure a separate log output that contains only packet drop and reject entries.

- **kea-dhcp4.callouts** - used to log messages pertaining to the callouts registration and execution for the particular hook point.

- **kea-dhcp4.commands** - used to log messages relating to the handling of commands received by the the DHCPv4 server over the command channel.

- **kea-dhcp4.ddns** - used by the DHCPv4 server to log messages related to the Client FQDN and Hostname option processing. It also includes log messages related to the relevant DNS updates.

- **kea-dhcp4.dhcp4** - used by the DHCPv4 server daemon to log basic operations.

- **kea-dhcp4.dhcpsrv** - the base logger for the libdhcpsrv library.

- **kea-dhcp4.eval** - used to log messages relating to the client classification expression evaluation code.

- **kea-dhcp4.hooks** - used to log messages related to management of hooks libraries, e.g. registration and deregistration of the libraries, and to the initialization of the callouts execution for various hook points within the DHCPv4 server.

- **kea-dhcp4.hosts** - used within the libdhcpsrv and it logs messages related to the management of the DHCPv4 host reservations, i.e. retrieval of the reservations and adding new reservations.

- **kea-dhcp4.leases** - used by the DHCPv4 server to log messages related to the lease allocation. The messages include detailed information about the allocated or offered leases, errors during the lease allocation etc.

- **kea-dhcp4.options** - used by the DHCPv4 server to log messages related to processing of the options in the DHCPv4 messages, i.e. parsing options, encoding options into on-wire format and packet classification using options contained in the received packets.

- **kea-dhcp4.packets** - this logger is mostly used to log messages related to transmission of the DHCPv4 packets, i.e. packet reception and sending a response. Such messages include information about the source and destination IP addresses and interfaces used to transmit packets. The logger is also used to log messages related to subnet selection, as this selection is usually based on the IP addresses and/or interface names, which can be retrieved from the received packet, even before the DHCPv4 message carried in the packet is parsed.

- **kea-dhcp6** - the root logger for the DHCPv6 server. All components used by the DHCPv6 server inherit the settings from this logger if there is no specialized logger provided.

- **kea-dhcp6.alloc-engine** - used used by the lease allocation engine, which is responsible for managing leases in the lease database, i.e. create, modify and remove DHCPv6 leases as a result of processing messages from the clients.

- **kea-dhcp6.bad-packets** - used used by the DHCPv6 server daemon for logging inbound client packets that were dropped.

- **kea-dhcp6.callouts** - used to log messages pertaining to the callouts registration and execution for the particular hook point.

- **kea-dhcp6.commands** - used to log messages relating to the handling of commands received by the the DHCPv6 server over the command channel.

- **kea-dhcp6.ddns** - this logger is used by the DHCPv6 server to log messages related to the Client FQDN option processing. It also includes log messages related to the relevant DNS updates.

- **kea-dhcp6.dhcp6** - used DHCPv6 server daemon to log basic operations.

- **kea-dhcp6.dhcpsrv** - the base logger for the libdhcpsrv library.

- **kea-dhcp6.eval** - used to log messages relating to the client classification expression evaluation code.

- **kea-dhcp6.hooks** - this logger is used to log messages related to management of hooks libraries, e.g. registration and deregistration of the libraries, and to the initialization of the callouts execution for various hook points within the DHCPv6 server.

- **kea-dhcp6.hosts** - used within the libdhcpsrv and it logs messages related to the management of the DHCPv6 host reservations, i.e. retrieval of the reservations and adding new reservations.

- **kea-dhcp6.leases** - used by the DHCPv6 server to log messages related to the lease allocation. The messages include detailed information about the allocated or offered leases, errors during the lease allocation etc.

- **kea-dhcp6.options** - used by the DHCPv6 server to log messages related to processing of the options in the DHCPv6 messages, i.e. parsing options, encoding options into on-wire format and packet classification using options contained in the received packets.

- **kea-dhcp6.packets** - this logger is mostly used to log messages related to transmission of the DHCPv6 packets, i.e. packet reception and sending a response. Such messages include the information about the source and destination IP addresses and interfaces used to transmit packets. This logger is also used to log messages related to subnet selection, as this selection is usually based on the IP addresses and/or interface names, which can be retrieved from the received packet, even before the DHCPv6 message carried in the packet is parsed.

- **kea-dhcp-ddns** - the root logger for the kea-dhcp-ddns daemon. All components used by this daemon inherit the settings from this logger if there is no specialized logger provided.

- **kea-dhcp-ddns.dhcpddns** - the logger used by the kea-dhcp-ddns daemon for logging configuration and global event information. This logger does not specify logging settings for libraries used by the daemon.

- **kea-dhcp-ddns.dhcp-to-d2** - used by the kea-dhcp-ddns daemon for logging information about events dealing with receiving messages from the DHCP servers and adding them to the queue for processing.

- **kea-dhcp-ddns.d2-to-dns** - used by the kea-dhcp-ddns daemon for logging information about events dealing with sending and receiving messages with the DNS servers.

Note that user-defined hook libraries should not use any of those loggers but should define new loggers with names that correspond to the libraries using them. Suppose that the user created the library called "libpacket-capture" to dump packets received and transmitted by the server to the file. The appropriate name for the logger could be **kea-dhcp4.packet-capture**. (Note that the hook library implementor only specifies the second part of this name, i.e. "packet-capture". The first part is a root logger name and is prepended by the Kea logging system.) It is also important to note that since this new logger is a child of a root logger, it inherits the configuration from the root logger, something that can be overridden by an entry in the configuration file.

The list of loggers above excludes any loggers implemented in hooks libraries. Please consult the documentation for the libraries for the names of the loggers they define.

Additional loggers may be defined in future versions of Kea. The easiest way to find out the logger name is to configure all logging to go to a single destination and look for specific logger names. See Section 17.1.2 for details.

17.1.1.2 severity (string)

This specifies the category of messages logged. Each message is logged with an associated severity which may be one of the following (in descending order of severity):

- FATAL - associated with messages generated by a condition that is so serious that the server cannot continue executing.

- ERROR- associated with messages generated by an error condition. The server will continue executing, but the results may not be as expected.

- WARN - indicates an out of the ordinary condition. However, the server will continue executing normally.

- INFO - an informational message marking some event.

- DEBUG - messages produced for debugging purposes.

When the severity of a logger is set to one of these values, it will only log messages of that severity and above (e.g. setting the logging severity to INFO will log INFO, WARN, ERROR and FATAL messages). The severity may also be set to NONE, in which case all messages from that logger are inhibited.

Note

The keactrl tool, described in Chapter 6, can be configured to start the servers in the verbose mode. If this is the case, the settings of the logging severity in the configuration file will have no effect, i.e. the servers will use logging severity of DEBUG regardless of the logging settings specified in the configuration file. If you need to control severity via configuration file, please make sure that the `kea_verbose` value is set to "no" within the keactrl configuration.

17.1.1.3 debuglevel (integer)

When a logger's severity is set to DEBUG, this value specifies what level of debug messages should be printed. It ranges from 0 (least verbose) to 99 (most verbose). If severity for the logger is not DEBUG, this value is ignored.

17.1.1.4 output_options (list)

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

17.1.1.4.1 output (string)

This value determines the type of output. There are several special values allowed here: **stdout** (messages are printed on standard output), **stderr** (messages are printed on stderr), **syslog** (messages are logged to syslog using default name, **syslog:name** (messages are logged to syslog using specified name). Any other value is interpreted as a filename to which messages should be written.

17.1.1.4.2 flush (true of false)

Flush buffers after each log message. Doing this will reduce performance but will ensure that if the program terminates abnormally, all messages up to the point of termination are output. The default is "true".

17.1.1.4.3 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 set to 0 or omitted, 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 version or later of log4cplus is used to build Kea, it should not happen. Even for older versions it is normally expected to happen rarely unless the log messages are produced very frequently by multiple different processes.

17.1.1.4.4 maxver (integer)

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

17.1.1.5 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"
        }
    ]
}
```

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,
                    "flush": true
                }
            ],
            "severity": "DEBUG",
            "debuglevel": 99
        }
    ]
}
```

17.1.2 Logging Message Format

Each message written to the configured logging destinations comprises a number of components that identify the origin of the message and, if the message indicates a problem, information about the problem that may be useful in fixing it.

Consider the message below logged to a file:

```
2014-04-11 12:58:01.005 INFO  [kea-dhcp4.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 elements: the Kea process generating the message (in this case, **kea-dhcp4**) and the component within the program from which the message originated (**dhcpsrv**, which is the name of the common library used by DHCP server implementations). The number after the slash is a process id (pid).

DHCPSRV_MEMFILE_DB
> The message identification. Every message in Kea has a unique identification, which can be used as an index into the *Kea Messages Manual* (`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.

17.1.3 Logging During Kea Startup

The logging configuration is specified in the configuration file. However, when Kea starts, the file is not read until some way into the initialization process. Prior to that, the logging settings are set to default values, although it is possible to modify some aspects of the settings by means of environment variables. Note that in the absence of any logging configuration in the configuration file, the settings of (possibly modified) default configuration will persist while the program is running.

The following environment variables can be used to control the behavior of logging during startup:

KEA_LOCKFILE_DIR

Specifies a directory where the logging system should create its lock file. If not specified, it is *prefix*/var/run/kea, where *prefix* defaults to /usr/local. This variable must not end with a slash. There is one special value: "none", which instructs Kea to not create lock file at all. This may cause issues if several processes log to the same file.

KEA_LOGGER_DESTINATION

Specifies logging output. There are several special values.

stdout

Log to standard output.

stderr

Log to standard error.

syslog[:*fac*]

Log via syslog. The optional *fac* (which is separated from the word "syslog" by a colon) specifies the facility to be used for the log messages. Unless specified, messages will be logged using the facility "local0".

Any other value is treated as a name of the output file. If not specified otherwise, Kea will log to standard output.

Chapter 18

Frequently Asked Questions

This chapter contains a number of frequently asked questions and troubleshooting tips. It currently lacks content, but it is expected to grow over time.

18.1 General Frequently Asked Questions

18.1.1 Where did the Kea name came from?

Kea is the name of a high mountain parrot living in New Zealand. See this https://lists.isc.org/pipermail/kea-users/2014-October/000032.html for an extended answer.

18.1.2 Feature X is not supported yet. When/if will it be available?

Kea is developed by a small team of engineers. Our resources are limited, so we need to prioritize requests. The complexity of a new feature (how difficult it is to implement a feature and how likely it would break something that already works), amount of work required and expected popularity (i.e., how many users would actually benefit from it) are three leading factors. We sometimes also have contractual obligations.

Simply stating that you'd like feature X is useful. We try to implement features that are actively requested first, but the reality is that we have more requests than we can handle, so some of them must be postponed, at least in the near future. So is your request likely to be rejected? Not at all. You can do many things to greatly improve the chances of your request being fulfilled. First, it helps to explain why you need a feature. If your explanation is reasonable and there are likely other users that would benefit from it, the chances for Kea developers to put this task on a roadmap is better. Saying that you are willing to participate in tests (e.g., test engineering drops when they become available) is also helpful.

Another thing you can do to greatly improve the chances of a feature to appear is to actually develop it on your own and submit a patch. That's an avenue that people often forget about. Kea is open source software and we do accept patches. There are certain requirements, like code quality, comments, unit-tests, documentation, etc., but we have accepted a significant number of patches in the past, so it's doable. Accepted contributions range from minor documentation corrections to significant new features, like support for a new database type. Before considering writing and submitting a patch, make sure you read the Contributor's Guide in the Kea Developer's Guide.

Kea is developed by ISC, which is a non-profit organization. You may consider signing a development contract with us. In the past we did implement certain features due to contractual obligations. With additional funds we are able to put extra engineering efforts into Kea development. We can reshuffle our schedule or add extra hands to the team if needed. Please keep in mind that Kea is open source software and its principle goal is to provide a good DHCP solution that can be used by everyone. In other words, we may refuse a contract that would tie the solution to specific proprietary technology or make it unusable for other users. Also, we strive to make Kea a reference implementation, so if your proposal significantly violates a RFC, we may have a problem with that. Nevertheless, please talk to us and we may be able to find a solution.

Finally, Kea has a public roadmap, with releases happening several times each year. We tend to not modify plans for the current milestone, unless there are very good reasons to do so. Therefore "I'd like a feature X in 6 months" is much better received than "I'd like a feature X now".

18.2 Frequently Asked Questions about DHCPv4

18.2.1 I set up a firewall, but the Kea server still receives the traffic. Why?

Any DHCPv4 server must be able to receive from and send traffic to hosts that don't have an IPv4 address assigned yet. That is typically not possible with regular UDP sockets, therefore the Kea DHCPv4 server uses raw sockets by default. Raw sockets mean that the incoming packets are received as raw Ethernet frames, thus bypassing the whole kernel IP stack, including any firewalling rules your kernel may provide.

If you do not want the server to use raw sockets, it is possible to configure the Kea DHCPv4 server to use UDP sockets instead. See **dhcp-socket-type** described in Section 7.2.4. However, using UDP sockets has certain limitations. In particular, they may not allow for sending responses directly to clients without IPv4 addresses assigned. That's ok, if all your traffic is coming through relay agents.

18.3 Frequently Asked Questions about DHCPv6

18.3.1 Kea DHCPv6 doesn't seem to get incoming traffic. I checked with tcpdump (or other traffic capture software) that the incoming traffic is reaching the box. What's wrong?

Please check whether your OS has any IPv6 filtering rules. Many operating systems are shipped with firewalls that discard incoming IPv6 traffic by default. In particular, many Linux distributions do that. Please check the output of the following command:

```
# ip6tables -L -n
```

One common mistake in this area is to use **iptables** tool, which lists IPv4 firewall rules only.

Chapter 19

Acknowledgments

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.