

BIND 10 Guide

Administrator Reference for BIND 10

Copyright © 2010-2012 Internet Systems Consortium, Inc.

Contents

1	Introduction	1
1.1	Supported Platforms	1
1.2	Required Software	1
1.3	Starting and Stopping the Server	1
1.4	Managing BIND 10	2
2	Installation	3
2.1	Building Requirements	3
2.2	Quick start	3
2.3	Installation from source	4
2.3.1	Download Tar File	4
2.3.2	Retrieve from Git	4
2.3.3	Configure before the build	5
2.3.4	Build	5
2.3.5	Install	5
2.3.6	Install Hierarchy	5
3	Starting BIND10 with bind10	6
3.1	Starting BIND 10	6
3.2	Configuration of started processes	6
4	Command channel	9
5	Configuration manager	10
6	Remote control daemon	11
6.1	Configuration specification for b10-cmdctl	11
7	Control and configure user interface	12
8	Authoritative Server	13
8.1	Server Configurations	13
8.2	Data Source Backends	14
8.2.1	In-memory Data Source	14
8.3	Loading Master Zones Files	14

9 Incoming Zone Transfers	16
9.1 Configuration for Incoming Zone Transfers	16
9.2 Enabling IXFR	16
9.3 Secondary Manager	17
9.4 Trigger an Incoming Zone Transfer Manually	17
10 Outbound Zone Transfers	18
11 Recursive Name Server	19
11.1 Access Control	19
11.2 Forwarding	20
12 DHCPv4 Server	21
12.1 DHCPv4 Server Usage	21
12.2 DHCPv4 Server Configuration	22
12.3 Supported standards	22
12.4 DHCPv4 Server Limitations	22
13 DHCPv6 Server	24
13.1 DHCPv6 Server Usage	24
13.2 DHCPv6 Server Configuration	25
13.3 Supported DHCPv6 Standards	25
13.4 DHCPv6 Server Limitations	25
14 libdhcp++ library	26
14.1 Interface detection	26
14.2 DHCPv4/DHCPv6 packet handling	26
15 Statistics	27
16 Logging	28
16.1 Logging configuration	28
16.1.1 Loggers	28
16.1.1.1 name (string)	28
16.1.1.2 severity (string)	29
16.1.1.3 output_options (list)	29
16.1.1.4 debuglevel (integer)	29
16.1.1.5 additive (true or false)	29
16.1.2 Output Options	29
16.1.2.1 destination (string)	29
16.1.2.2 output (string)	30
16.1.2.2.1 flush (true of false)	30

16.1.2.2.2	maxsize (integer)	30
16.1.2.2.3	maxver (integer)	30
16.1.3	Example session	30
16.2	Logging Message Format	32

List of Tables

Abstract

BIND 10 is a framework that features Domain Name System (DNS) suite and Dynamic Host Configuration Protocol (DHCP) servers managed by Internet Systems Consortium (ISC). It includes DNS libraries, modular components for controlling authoritative and recursive DNS servers, and experimental DHCPv4 and DHCPv6 servers.

This is the reference guide for BIND 10 version 20120329. The most up-to-date version of this document (in PDF, HTML, and plain text formats), along with other documents for BIND 10, can be found at <http://bind10.isc.org/docs>.

Preface

Acknowledgements

ISC would like to acknowledge generous support for BIND 10 development of DHCPv4 and DHCPv6 components provided by [Comcast](#).

Chapter 1

Introduction

BIND is the popular implementation of a DNS server, developer interfaces, and DNS tools. BIND 10 is a rewrite of BIND 9. BIND 10 is written in C++ and Python and provides a modular environment for serving and maintaining DNS. BIND 10 provides a EDNS0- and DNSSEC-capable authoritative DNS server and a caching recursive name server which also provides forwarding.

This guide covers the experimental prototype of BIND 10 version 20120329.

1.1 Supported Platforms

BIND 10 builds have been tested on Debian GNU/Linux 5 and unstable, Ubuntu 9.10, NetBSD 5, Solaris 10, FreeBSD 7 and 8, CentOS Linux 5.3, and MacOS 10.6. It has been tested on Sparc, i386, and amd64 hardware platforms. It is planned for BIND 10 to build, install and run on Windows and standard Unix-type platforms.

1.2 Required Software

BIND 10 requires at least Python 3.1 (<http://www.python.org/>). It has also been tested with Python 3.2.

BIND 10 uses the Botan crypto library for C++ (<http://botan.randombit.net/>). It requires at least Botan version 1.8.

BIND 10 uses the log4cplus C++ logging library (<http://log4cplus.sourceforge.net/>). It requires at least log4cplus version 1.0.3.

The authoritative DNS server uses SQLite3 (<http://www.sqlite.org/>). It needs at least SQLite version 3.3.9.

The **b10-xfrin**, **b10-xfrout**, and **b10-zonemgr** components require the libpython3 library and the Python `_sqlite3.so` module (which is included with Python). The Python module needs to be built for the corresponding Python 3.

Note

Some operating systems do not provide these dependencies in their default installation nor standard packages collections. You may need to install them separately.

1.3 Starting and Stopping the Server

BIND 10 is modular. Part of this modularity is accomplished using multiple cooperating processes which, together, provide the server functionality. This is a change from the previous generation of BIND software, which used a single process.

At first, running many different processes may seem confusing. However, these processes are started, stopped, and maintained by a single command, **bind10**. This command starts a master process which will start other processes as needed. The processes started by the **bind10** command have names starting with "b10-", including:

- **b10-auth** — Authoritative DNS server. This process serves DNS requests.
- **b10-cfgmgr** — Configuration manager. This process maintains all of the configuration for BIND 10.
- **b10-cmdctl** — Command and control service. This process allows external control of the BIND 10 system.
- **b10-msgq** — Message bus daemon. This process coordinates communication between all of the other BIND 10 processes.
- **b10-resolver** — Recursive name server. This process handles incoming queries.
- **b10-sockcreator** — Socket creator daemon. This process creates sockets used by network-listening BIND 10 processes.
- **b10-stats** — Statistics collection daemon. This process collects and reports statistics data.
- **b10-stats-httpd** — HTTP server for statistics reporting. This process reports statistics data in XML format over HTTP.
- **b10-xfrin** — Incoming zone transfer service. This process is used to transfer a new copy of a zone into BIND 10, when acting as a secondary server.
- **b10-xfrout** — Outgoing zone transfer service. This process is used to handle transfer requests to send a local zone to a remote secondary server, when acting as a master server.
- **b10-zonemgr** — Secondary manager. This process keeps track of timers and other necessary information for BIND 10 to act as a slave server.

These are ran automatically by **bind10** and do not need to be run manually.

1.4 Managing BIND 10

Once BIND 10 is running, a few commands are used to interact directly with the system:

- **bindctl** — interactive administration interface. This is a low-level command-line tool which allows a developer or an experienced administrator to control BIND 10.
- **b10-loadzone** — zone file loader. This tool will load standard masterfile-format zone files into BIND 10.
- **b10-cmdctl-usermgr** — user access control. This tool allows an administrator to authorize additional users to manage BIND 10.

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

BIND 10 also provides libraries and programmer interfaces for C++ and Python for the message bus, configuration backend, and, of course, DNS. These include detailed developer documentation and code examples.

Chapter 2

Installation

2.1 Building Requirements

In addition to the run-time requirements, building BIND 10 from source code requires various development include headers.

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 BIND 10 from source code.

Building from source code requires the Boost build-time headers (<http://www.boost.org/>). At least Boost version 1.35 is required.

To build BIND 10, also install the Botan (at least version 1.8) and the log4cplus (at least version 1.0.3) development include headers.

Building BIND 10 also requires a C++ compiler and standard development headers, make, and pkg-config. BIND 10 builds have been tested with GCC g++ 3.4.3, 4.1.2, 4.1.3, 4.2.1, 4.3.2, and 4.4.1; Clang++ 2.8; and Sun C++ 5.10.

Visit the wiki at <http://bind10.isc.org/wiki/SystemSpecificNotes> for system-specific installation tips.

2.2 Quick start

Note

This quickly covers the standard steps for installing and deploying BIND 10 as an authoritative name server using its defaults. For troubleshooting, full customizations and further details, see the respective chapters in the BIND 10 guide.

To quickly get started with BIND 10, follow these steps.

1. Install required run-time and build dependencies.
2. Download the BIND 10 source tar file from <ftp://ftp.isc.org/isc/bind10/>.
3. Extract the tar file:

```
$ gzcat bind10-VERSION.tar.gz | tar -xvf -
```

4. Go into the source and run configure:
-

```
$ cd bind10-VERSION
$ ./configure
```

5. Build it:

```
$ make
```

6. Install it (to default /usr/local):

```
$ make install
```

7. Start the server:

```
$ /usr/local/sbin/bind10
```

8. Test it; for example:

```
$ dig @127.0.0.1 -c CH -t TXT authors.bind
```

9. Load desired zone file(s), for example:

```
$ b10-loadzone your.zone.example.org
```

10. Test the new zone.

2.3 Installation from source

BIND 10 is open source software written in C++ and Python. It is freely available in source code form from ISC via the Git code revision control system or as a downloadable tar file. It may also be available in pre-compiled ready-to-use packages from operating system vendors.

2.3.1 Download Tar File

Downloading a release tar file is the recommended method to obtain the source code.

The BIND 10 releases are available as tar file downloads from <ftp://ftp.isc.org/isc/bind10/>. Periodic development snapshots may also be available.

2.3.2 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 using source code retrieved via Git additional software will be required: automake (v1.11 or newer), libtoolize, and autoconf (2.59 or newer). These may need to be installed.

The latest development code, including temporary experiments and un-reviewed code, is available via the BIND 10 code revision control system. This is powered by Git and all the BIND 10 development is public. The leading development is done in the 'master'.

The code can be checked out from <git://git.bind10.isc.org/bind10>; for example:

```
$ git clone git://git.bind10.isc.org/bind10
```

When checking out the code from the code version control system, it doesn't include the generated configure script, Makefile.in files, nor the related configure files. They can be created by running **autoreconf** with the **--install** switch. This will run **autoconf**, **aclocal**, **libtoolize**, **autoheader**, **automake**, and related commands.

2.3.3 Configure before the build

BIND 10 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. The 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-pythonpath Define the path to Python 3.1 if it is not in the standard execution path.

--with-gtest Enable building the C++ Unit Tests using the Google Tests framework. Optionally this can define the path to the gtest header files and library.

For example, the following configures it to find the Boost headers, find the Python interpreter, and sets the installation location:

```
$ ./configure \
    --with-boost-include=/usr/pkg/include \
    --with-pythonpath=/usr/pkg/bin/python3.1 \
    --prefix=/opt/bind10
```

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

2.3.4 Build

After the configure step is complete, to build the executables from the C++ code and prepare the Python scripts, run:

```
$ make
```

2.3.5 Install

To install the BIND 10 executables, support files, and documentation, run:

```
$ make install
```

Note

The install step may require superuser privileges.

2.3.6 Install Hierarchy

The following is the layout of the complete BIND 10 installation:

- `bin/` — general tools and diagnostic clients.
 - `etc/bind10-devel/` — configuration files.
 - `lib/` — libraries and python modules.
 - `libexec/bind10-devel/` — executables that a user wouldn't normally run directly and are not run independently. These are the BIND 10 modules which are daemons started by the **bind10** tool.
 - `sbin/` — commands used by the system administrator.
 - `share/bind10-devel/` — configuration specifications.
 - `share/man/` — manual pages (online documentation).
 - `var/bind10-devel/` — data source and configuration databases.
-

Chapter 3

Starting BIND10 with bind10

BIND 10 provides the **bind10** command which starts up the required processes. **bind10** will also restart some processes that exit unexpectedly. This is the only command needed to start the BIND 10 system.

After starting the **b10-msgq** communications channel, **bind10** connects to it, runs the configuration manager, and reads its own configuration. Then it starts the other modules.

The **b10-sockcreator**, **b10-msgq** and **b10-cfgmgr** services make up the core. The **b10-msgq** daemon provides the communication channel between every part of the system. The **b10-cfgmgr** daemon is always needed by every module, if only to send information about themselves somewhere, but more importantly to ask about their own settings, and about other modules. The **b10-sockcreator** will allocate sockets for the rest of the system.

In its default configuration, the **bind10** master process will also start up **b10-cmdctl** for administration tools to communicate with the system, **b10-auth** for authoritative DNS service, **b10-stats** for statistics collection, **b10-stats-httpd** for statistics reporting, **b10-xfrin** for inbound DNS zone transfers, **b10-xfrout** for outbound DNS zone transfers, and **b10-zonemgr** for secondary service.

3.1 Starting BIND 10

To start the BIND 10 service, simply run **bind10**. Run it with the `--verbose` switch to get additional debugging or diagnostic output.

Note

If the `setproctitle` Python module is detected at start up, the process names for the Python-based daemons will be renamed to better identify them instead of just 'python'. This is not needed on some operating systems.

3.2 Configuration of started processes

The processes to be started can be configured, with the exception of the **b10-sockcreator**, **b10-msgq** and **b10-cfgmgr**.

The configuration is in the `Boss/components` section. Each element represents one component, which is an abstraction of a process (currently there's also one component which doesn't represent a process). If you didn't want to transfer out at all (your server is a slave only), you would just remove the corresponding component from the set, like this and the process would be stopped immediately (and not started on the next startup):

```
> config remove Boss/components b10-xfrout
> config commit
```

To add a process to the set, let's say the resolver (which not started by default), you would do this:

```
> config add Boss/components b10-resolver
> config set Boss/components/b10-resolver/special resolver
> config set Boss/components/b10-resolver/kind needed
> config set Boss/components/b10-resolver/priority 10
> config commit
```

Now, what it means. We add an entry called `b10-resolver`. It is both a name used to reference this component in the configuration and the name of the process to start. Then we set some parameters on how to start it.

The special one is for components that need some kind of special care during startup or shutdown. Unless specified, the component is started in usual way. This is the list of components that need to be started in a special way, with the value of special used for them:

Component	Special	Description
b10-auth	auth	Authoritative server
b10-resolver	resolver	The resolver
b10-cmdctl	cmdctl	The command control (remote control interface)

The kind specifies how a failure of the component should be handled. If it is set to ‘dispensable’ (the default unless you set something else), it will get started again if it fails. If it is set to ‘needed’ and it fails at startup, the whole **bind10** shuts down and exits with error exit code. But if it fails some time later, it is just started again. If you set it to ‘core’, you indicate that the system is not usable without the component and if such component fails, the system shuts down no matter when the failure happened. This is the behaviour of the core components (the ones you can’t turn off), but you can declare any other components as core as well if you wish (but you can turn these off, they just can’t fail).

The priority defines order in which the components should start. The ones with higher number are started sooner than the ones with lower ones. If you don’t set it, 0 (zero) is used as the priority. Usually, leaving it at the default is enough.

There are other parameters we didn’t use in our example. One of them is ‘address’. It is the address used by the component on the **b10-msgq** message bus. The special components already know their address, but the usual ones don’t. The address is by convention the thing after *b10-*, with the first letter capital (eg. **b10-stats** would have ‘Stats’ as its address).

The last one is process. It is the name of the process to be started. It defaults to the name of the component if not set, but you can use this to override it.

Note

This system allows you to start the same component multiple times (by including it in the configuration with different names, but the same process setting). However, the rest of the system doesn’t expect such a situation, so it would probably not do what you want. Such support is yet to be implemented.

Note

The configuration is quite powerful, but that includes a lot of space for mistakes. You could turn off the **b10-cmdctl**, but then you couldn’t change it back the usual way, as it would require it to be running (you would have to find and edit the configuration directly). Also, some modules might have dependencies: **b10-stats-httpd** needs **b10-stats**, **b10-xfrout** needs **b10-auth** to be running, etc.

In short, you should think twice before disabling something here.

It is possible to start some components multiple times (currently **b10-auth** and **b10-resolzer**). You might want to do that to gain more performance (each one uses only single core). Just put multiple entries under different names, like this, with the same config:

```
> config add Boss/components b10-resolver-2
> config set Boss/components/b10-resolver-2/special resolver
> config set Boss/components/b10-resolver-2/kind needed
> config commit
```

However, this is work in progress and the support is not yet complete. For example, each resolver will have its own cache, each authoritative server will keep its own copy of in-memory data and there could be problems with locking the sqlite database, if used. The configuration might be changed to something more convenient in future.

Chapter 4

Command channel

The BIND 10 components use the **b10-msgq** message routing daemon to communicate with other BIND 10 components. The **b10-msgq** implements what is called the 'Command Channel'. Processes intercommunicate by sending messages on the command channel. Example messages include shutdown, get configurations, and set configurations. This Command Channel is not used for DNS message passing. It is used only to control and monitor the BIND 10 system.

Administrators do not communicate directly with the **b10-msgq** daemon. By default, BIND 10 uses port 9912 for the **b10-msgq** service. It listens on 127.0.0.1.

Chapter 5

Configuration manager

The configuration manager, **b10-cfgmgr**, handles all BIND 10 system configuration. It provides persistent storage for configuration, and notifies running modules of configuration changes.

The **b10-auth** and **b10-xfrin** daemons and other components receive their configurations from the configuration manager over the **b10-msgq** command channel.

The administrator doesn't connect to it directly, but uses a user interface to communicate with the configuration manager via **b10-cmdctl**'s REST-ful interface. **b10-cmdctl** is covered in Chapter 6.

Note

The development prototype release only provides **bindctl** as a user interface to **b10-cmdctl**. Upcoming releases will provide another interactive command-line interface and a web-based interface.

The **b10-cfgmgr** daemon can send all specifications and all current settings to the **bindctl** client (via **b10-cmdctl**).

b10-cfgmgr relays configurations received from **b10-cmdctl** to the appropriate modules.

The stored configuration file is at `/usr/local/var/bind10-devel/b10-config.db`. (The full path is what was defined at build configure time for `--localstatedir`. The default is `/usr/local/var/`.) The format is loosely based on JSON and is directly parseable python, but this may change in a future version. This configuration data file is not manually edited by the administrator.

The configuration manager does not have any command line arguments. Normally it is not started manually, but is automatically started using the **bind10** master process (as covered in Chapter 3).

Chapter 6

Remote control daemon

b10-cmdctl is the gateway between administrators and the BIND 10 system. It is a HTTPS server that uses standard HTTP Digest Authentication for username and password validation. It provides a REST-ful interface for accessing and controlling BIND 10.

When **b10-cmdctl** starts, it firsts asks **b10-cfgmgr** about what modules are running and what their configuration is (over the **b10-msgq** channel). Then it will start listening on HTTPS for clients — the user interface — such as **bindctl**.

b10-cmdctl directly sends commands (received from the user interface) to the specified component. Configuration changes are actually commands to **b10-cfgmgr** so are sent there.

The HTTPS server requires a private key, such as a RSA PRIVATE KEY. The default location is at `/usr/local/etc/bind10-devel/cmdctl-keyfile.pem`. (A sample key is at `/usr/local/share/bind10-devel/cmdctl-keyfile.pem`.) It also uses a certificate located at `/usr/local/etc/bind10-devel/cmdctl-certfile.pem`. (A sample certificate is at `/usr/local/share/bind10-devel/cmdctl-certfile.pem`.) This may be a self-signed certificate or purchased from a certification authority.

Note

The HTTPS server doesn't support a certificate request from a client (at this time). The **b10-cmdctl** daemon does not provide a public service. If any client wants to control BIND 10, then a certificate needs to be first received from the BIND 10 administrator. The BIND 10 installation provides a sample PEM bundle that matches the sample key and certificate.

The **b10-cmdctl** daemon also requires the user account file located at `/usr/local/etc/bind10-devel/cmdctl-accounts.csv`. This comma-delimited file lists the accounts with a user name, hashed password, and salt. (A sample file is at `/usr/local/share/bind10-devel/cmdctl-accounts.csv`. It contains the user named 'root' with the password 'bind10'.)

The administrator may create a user account with the **b10-cmdctl-usermgr** tool.

By default the HTTPS server listens on the localhost port 8080. The port can be set by using the `--port` command line option. The address to listen on can be set using the `--address` command line argument. Each HTTPS connection is stateless and times out in 1200 seconds by default. This can be redefined by using the `--idle-timeout` command line argument.

6.1 Configuration specification for b10-cmdctl

The configuration items for **b10-cmdctl** are: `key_file` `cert_file` `accounts_file`

The control commands are: `print_settings` `shutdown`

Chapter 7

Control and configure user interface

Note

For this development prototype release, **bindctl** is the only user interface. It is expected that upcoming releases will provide another interactive command-line interface and a web-based interface for controlling and configuring BIND 10.

The **bindctl** tool provides an interactive prompt for configuring, controlling, and querying the BIND 10 components. It communicates directly with a REST-ful interface over HTTPS provided by **b10-cmdctl**. It doesn't communicate to any other components directly.

Configuration changes are actually commands to **b10-cfgmgr**. So when **bindctl** sends a configuration, it is sent to **b10-cmdctl** (over a HTTPS connection); then **b10-cmdctl** sends the command (over a **b10-msgq** command channel) to **b10-cfgmgr** which then stores the details and relays (over a **b10-msgq** command channel) the configuration on to the specified module.

Chapter 8

Authoritative Server

The **b10-auth** is the authoritative DNS server. It supports EDNS0 and DNSSEC. It supports IPv6. Normally it is started by the **bind10** master process.

8.1 Server Configurations

b10-auth is configured via the **b10-cfgmgr** configuration manager. The module name is 'Auth'. The configuration data items are:

database_file This is an optional string to define the path to find the SQLite3 database file. Note: Later the DNS server will use various data source backends. This may be a temporary setting until then.

datasources `datasources` configures data sources. The list items include: `type` to define the required data source type (such as 'memory'); `class` to optionally select the class (it defaults to 'IN'); and `zones` to define the file path name and the `origin` (default domain). By default, this is empty.

Note

In this development version, currently this is only used for the memory data source. Only the IN class is supported at this time. By default, the memory data source is disabled. Also, currently the zone file must be canonical such as generated by **named-compilezone -D**.

listen_on `listen_on` is a list of addresses and ports for **b10-auth** to listen on. The list items are the `address` string and `port` number. By default, **b10-auth** listens on port 53 on the IPv6 (::) and IPv4 (0.0.0.0) wildcard addresses.

statistics-interval `statistics-interval` is the timer interval in seconds for **b10-auth** to share its statistics information to `b10-stats(8)`. Statistics updates can be disabled by setting this to 0. The default is 60.

The configuration commands are:

loadzone `loadzone` tells **b10-auth** to load or reload a zone file. The arguments include: `class` which optionally defines the class (it defaults to 'IN'); `origin` is the domain name of the zone; and `datasrc` optionally defines the type of datasource (it defaults to 'memory').

Note

In this development version, currently this only supports the IN class and the memory data source.

sendstats `sendstats` tells **b10-auth** to send its statistics data to `b10-stats(8)` immediately.

shutdown Stop the authoritative DNS server. This has an optional `pid` argument to select the process ID to stop. (Note that the BIND 10 boss process may restart this service if configured.)

8.2 Data Source Backends

Note

For the development prototype release, **b10-auth** supports a SQLite3 data source backend and in-memory data source backend. Upcoming versions will be able to use multiple different data sources, such as MySQL and Berkeley DB.

By default, the SQLite3 backend uses the data file located at `/usr/local/var/bind10-devel/zone.sqlite3`. (The full path is what was defined at build configure time for `--localstatedir`. The default is `/usr/local/var/`.) This data file location may be changed by defining the `'database_file'` configuration.

8.2.1 In-memory Data Source

The following commands to **bindctl** provide an example of configuring an in-memory data source containing the `'example.com'` zone with the zone file named `'example.com.zone'`:

```
> config add Auth/datasources
> config set Auth/datasources[0]/type "memory"
> config add Auth/datasources[0]/zones
> config set Auth/datasources[0]/zones[0]/origin "example.com"
> config set Auth/datasources[0]/zones[0]/file "example.com.zone"
> config commit
```

The authoritative server will begin serving it immediately after it is loaded.

Use the **Auth loadzone** command in **bindctl** to reload a changed master file into memory; for example:

```
> Auth loadzone origin="example.com"
```

By default, the memory data source is disabled; it must be configured explicitly. To disable all the in-memory zones, specify a null list for `Auth/datasources`:

```
> config set Auth/datasources/ []
> config commit
```

The following example stops serving a specific zone:

```
> config remove Auth/datasources[0]/zones[0]
> config commit
```

(Replace the list number(s) in `datasources[0]` and/or `zones[0]` for the relevant zone as needed.)

8.3 Loading Master Zones Files

RFC 1035 style DNS master zone files may imported into a BIND 10 SQLite3 data source by using the **b10-loadzone** utility.

b10-loadzone supports the following special directives (control entries):

\$INCLUDE Loads an additional zone file. This may be recursive.

\$ORIGIN Defines the relative domain name.

\$TTL Defines the time-to-live value used for following records that don't include a TTL.

The `-o` argument may be used to define the default origin for loaded zone file records.

Note

In the development prototype release, only the SQLite3 back end is used by **b10-loadzone**. By default, it stores the zone data in `/usr/local/var/bind10-devel/zone.sqlite3` unless the `-d` switch is used to set the database filename. Multiple zones are stored in a single SQLite3 zone database.

If you reload a zone already existing in the database, all records from that prior zone disappear and a whole new set appears.

Chapter 9

Incoming Zone Transfers

Incoming zones are transferred using the **b10-xfrin** process which is started by **bind10**. When received, the zone is stored in the corresponding BIND 10 data source, and its records can be served by **b10-auth**. In combination with **b10-zonemgr** (for automated SOA checks), this allows the BIND 10 server to provide ‘secondary’ service.

The **b10-xfrin** process supports both AXFR and IXFR. Due to some implementation limitations of the current development release, however, it only tries AXFR by default, and care should be taken to enable IXFR.

Note

In the current development release of BIND 10, incoming zone transfers are only available for SQLite3-based data sources, that is, they don't work for an in-memory data source.

9.1 Configuration for Incoming Zone Transfers

In practice, you need to specify a list of secondary zones to enable incoming zone transfers for these zones (you can still trigger a zone transfer manually, without a prior configuration (see below)).

For example, to enable zone transfers for a zone named "example.com" (whose master address is assumed to be 2001:db8::53 here), run the following at the **bindctl** prompt:

```
> config add Xfrin/zones
> config set Xfrin/zones[0]/name "example.com"
> config set Xfrin/zones[0]/master_addr "2001:db8::53"
> config commit
```

(We assume there has been no zone configuration before).

9.2 Enabling IXFR

As noted above, **b10-xfrin** uses AXFR for zone transfers by default. To enable IXFR for zone transfers for a particular zone, set the **use_ixfr** configuration parameter to **true**. In the above example of configuration sequence, you'll need to add the following before performing **commit**:

```
> config set Xfrin/zones[0]/use_ixfr true
```

Note

One reason why IXFR is disabled by default in the current release is because it does not support automatic fallback from IXFR to AXFR when it encounters a primary server that doesn't support outbound IXFR (and, not many existing implementations support it). Another, related reason is that it does not use AXFR even if it has no knowledge about the zone (like at the very first time the secondary server is set up). IXFR requires the "current version" of the zone, so obviously it doesn't work in this situation and AXFR is the only workable choice. The current release of **b10-xfrin** does not make this selection automatically. These features will be implemented in a near future version, at which point we will enable IXFR by default.

9.3 Secondary Manager

The **b10-zonemgr** process is started by **bind10**. It keeps track of SOA refresh, retry, and expire timers and other details for BIND 10 to perform as a slave. When the **b10-auth** authoritative DNS server receives a NOTIFY message, **b10-zonemgr** may tell **b10-xfrin** to do a refresh to start an inbound zone transfer. The secondary manager resets its counters when a new zone is transferred in.

Note

Access control (such as allowing notifies) is not yet provided. The primary/secondary service is not yet complete.

The following example shows using **bindctl** to configure the server to be a secondary for the example zone:

```
> config add Zonemgr/secondary_zones
> config set Zonemgr/secondary_zones[0]/name "example.com"
> config set Zonemgr/secondary_zones[0]/class "IN"
> config commit
```

If the zone does not exist in the data source already (i.e. no SOA record for it), **b10-zonemgr** will automatically tell **b10-xfrin** to transfer the zone in.

9.4 Trigger an Incoming Zone Transfer Manually

To manually trigger a zone transfer to retrieve a remote zone, you may use the **bindctl** utility. For example, at the **bindctl** prompt run:

```
> Xfrin retransfer zone_name="foo.example.org" master=192.0.2.99
```

Chapter 10

Outbound Zone Transfers

The **b10-xfrout** process is started by **bind10**. When the **b10-auth** authoritative DNS server receives an AXFR or IXFR request, **b10-auth** internally forwards the request to **b10-xfrout**, which handles the rest of request processing. This is used to provide primary DNS service to share zones to secondary name servers. The **b10-xfrout** is also used to send NOTIFY messages to secondary servers.

A global or per zone `transfer_acl` configuration can be used to control accessibility of the outbound zone transfer service. By default, **b10-xfrout** allows any clients to perform zone transfers for any zones:

```
> config show Xfrout/transfer_acl
Xfrout/transfer_acl[0] {"action": "ACCEPT"} any (default)
```

You can change this to, for example, rejecting all transfer requests by default while allowing requests for the transfer of zone "example.com" from 192.0.2.1 and 2001:db8::1 as follows:

```
> config set Xfrout/transfer_acl[0] {"action": "REJECT"}
> config add Xfrout/zone_config
> config set Xfrout/zone_config[0]/origin "example.com"
> config set Xfrout/zone_config[0]/transfer_acl [{"action": "ACCEPT", "from": "192.0.2.1"},
                                                {"action": "ACCEPT", "from": "2001:db8 ←
::1"}]}
> config commit
```

Note

In the above example the lines for `transfer_acl` were divided for readability. In the actual input it must be in a single line.

If you want to require TSIG in access control, a system wide TSIG "key ring" must be configured. For example, to change the previous example to allowing requests from 192.0.2.1 signed by a TSIG with a key name of "key.example", you'll need to do this:

```
> config set tsig_keys/keys ["key.example:<base64-key>"]
> config set Xfrout/zone_config[0]/transfer_acl [{"action": "ACCEPT", "from": "192.0.2.1", ←
"key": "key.example"}]}
> config commit
```

Both Xfrout and Auth will use the system wide keyring to check TSIGs in the incoming messages and to sign responses.

Note

The way to specify zone specific configuration (ACLs, etc) is likely to be changed.

Chapter 11

Recursive Name Server

The **b10-resolver** process is started by **bind10**.

The main **bind10** process can be configured to select to run either the authoritative or resolver or both. By default, it starts the authoritative service. You may change this using **bindctl**, for example:

```
> config remove Boss/components b10-xfrout
> config remove Boss/components b10-xfrin
> config remove Boss/components b10-auth
> config add Boss/components b10-resolver
> config set Boss/components/b10-resolver/special resolver
> config set Boss/components/b10-resolver/kind needed
> config set Boss/components/b10-resolver/priority 10
> config commit
```

The master **bind10** will stop and start the desired services.

By default, the resolver listens on port 53 for 127.0.0.1 and ::1. The following example shows how it can be configured to listen on an additional address (and port):

```
> config add Resolver/listen_on
> config set Resolver/listen_on[2]/address "192.168.1.1"
> config set Resolver/listen_on[2]/port 53
> config commit
```

(Replace the '2' as needed; run 'config show Resolver/listen_on' if needed.)

11.1 Access Control

By default, the **b10-resolver** daemon only accepts DNS queries from the localhost (127.0.0.1 and ::1). The `Resolver/query_acl` configuration may be used to reject, drop, or allow specific IPs or networks. This configuration list is first match.

The configuration's `action` item may be set to 'ACCEPT' to allow the incoming query, 'REJECT' to respond with a DNS REFUSED return code, or 'DROP' to ignore the query without any response (such as a blackhole). For more information, see the respective debugging messages: [RESOLVER_QUERY_ACCEPTED](#), [RESOLVER_QUERY_REJECTED](#), and [RESOLVER_QUERY_DENIED](#).

The required configuration's `from` item is set to an IPv4 or IPv6 address, addresses with an network mask, or to the special lowercase keywords 'any6' (for any IPv6 address) or 'any4' (for any IPv4 address).

For example to allow the `192.168.1.0/24` network to use your recursive name server, at the **bindctl** prompt run:

```
> config add Resolver/query_acl
> config set Resolver/query_acl[2]/action "ACCEPT"
> config set Resolver/query_acl[2]/from "192.168.1.0/24"
> config commit
```

(Replace the '2' as needed; run `config show Resolver/query_acl` if needed.)

Note

This prototype access control configuration syntax may be changed.

11.2 Forwarding

To enable forwarding, the upstream address and port must be configured to forward queries to, such as:

```
> config set Resolver/forward_addresses [{ "address": "192.168.1.1", "port": 53 }]
> config commit
```

(Replace `192.168.1.1` to point to your full resolver.)

Normal iterative name service can be re-enabled by clearing the forwarding address(es); for example:

```
> config set Resolver/forward_addresses []
> config commit
```

Chapter 12

DHCPv4 Server

Dynamic Host Configuration Protocol for IPv4 (DHCP or DHCPv4) and Dynamic Host Configuration Protocol for IPv6 (DHCPv6) are protocols that allow one node (server) to provision configuration parameters to many hosts and devices (clients). To ease deployment in larger networks, additional nodes (relays) may be deployed that facilitate communication between servers and clients. Even though principles of both DHCPv4 and DHCPv6 are somewhat similar, these are two radically different protocols. BIND10 offers server implementations for both DHCPv4 and DHCPv6. This chapter is about DHCP for IPv4. For a description of the DHCPv6 server, see Chapter 13.

The DHCPv4 server component is currently under intense development. You may want to check out [BIND10 DHCP \(Kea\) wiki](#) and recent posts on [BIND10 developers mailing list](#).

The DHCPv4 and DHCPv6 components in BIND10 architecture are internally code named ‘Kea’.

Note

As of December 2011, both DHCPv4 and DHCPv6 components are skeleton servers. That means that while they are capable of performing DHCP configuration, they are not fully functional yet. In particular, neither has functional lease databases. This means that they will assign the same, fixed, hardcoded addresses to any client that will ask. See Section 12.4 and Section 13.4 for detailed description.

12.1 DHCPv4 Server Usage

BIND10 provides the DHCPv4 server component since December 2011. It is a skeleton server and can be described as an early prototype that is not fully functional yet. It is mature enough to conduct first tests in lab environment, but it has significant limitations. See Section 12.4 for details.

The DHCPv4 server is implemented as **b10-dhcp4** daemon. As it is not configurable yet, it is fully autonomous, that is it does not interact with **b10-cfgmgr**. To start DHCPv4 server, simply input:

```
#cd src/bin/dhcp4
#./b10-dhcp4
```

Depending on your installation, **b10-dhcp4** binary may reside in `src/bin/dhcp4` in your source code directory, in `/usr/local/bin/b10-dhcp4` or other directory you specified during compilation. At start, the server will detect available network interfaces and will attempt to open UDP sockets on all interfaces that are up, running, are not loopback, and have IPv4 address assigned. The server will then listen to incoming traffic. Currently supported client messages are DISCOVER and REQUEST. The server will respond to them with OFFER and ACK, respectively. Since the DHCPv4 server opens privileged ports, it requires root access. Make sure you run this daemon as root.

Note

Integration with **bind10** is planned. Ultimately, **b10-dhcp4** will not be started directly, but rather via **bind10**. Please be aware of this planned change.

12.2 DHCPv4 Server Configuration

The DHCPv4 server does not have a lease database implemented yet nor any support for configuration, so every time the same set of configuration options (including the same fixed address) will be assigned every time.

At this stage of development, the only way to alter the server configuration is to tweak its source code. To do so, please edit `src/bin/dhcp4/dhcp4_srv.cc` file and modify following parameters and recompile:

```
const std::string HARDCODED_LEASE = "192.0.2.222"; // assigned lease
const std::string HARDCODED_NETMASK = "255.255.255.0";
const uint32_t HARDCODED_LEASE_TIME = 60; // in seconds
const std::string HARDCODED_GATEWAY = "192.0.2.1";
const std::string HARDCODED_DNS_SERVER = "192.0.2.2";
const std::string HARDCODED_DOMAIN_NAME = "isc.example.com";
const std::string HARDCODED_SERVER_ID = "192.0.2.1";
```

Lease database and configuration support is planned for 2012.

12.3 Supported standards

The following standards and draft standards are currently supported:

- RFC2131: Supported messages are DISCOVER, OFFER, REQUEST, and ACK.
- RFC2132: 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).

12.4 DHCPv4 Server Limitations

These are the current limitations of the DHCPv4 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.

- During initial IPv4 node configuration, the server is expected to send packets to a node that does not have IPv4 address assigned yet. The server requires certain tricks (or hacks) to transmit such packets. This is not implemented yet, therefore DHCPv4 server supports relayed traffic only (that is, normal point to point communication).
- **b10-dhcp4** provides a single, fixed, hardcoded lease to any client that asks. There is no lease manager implemented. If two clients request addresses, they will both get the same fixed address.
- **b10-dhcp4** does not support any configuration mechanisms yet. The whole configuration is currently hardcoded. The only way to tweak configuration is to directly modify source code. See Section 12.2 for details.
- Upon start, the server will open sockets on all interfaces that are not loopback, are up and running and have IPv4 address. Support for multiple interfaces is not coded in reception routines yet, so if you are running this code on a machine that has many interfaces and **b10-dhcp4** happens to listen on wrong interface, the easiest way to work around this problem is to turn down other interfaces. This limitation will be fixed shortly.
- PRL (Parameter Request List, a list of options requested by a client) is currently ignored and server assigns DNS SERVER and DOMAIN NAME options.
- **b10-dhcp4** does not support BOOTP. That is a design choice. This limitation is permanent. If you have legacy nodes that can't use DHCP and require BOOTP support, please use latest version of ISC DHCP <http://www.isc.org/software/dhcp>.
- Interface detection is currently working on Linux only. See Section 14.1 for details.

- **b10-dhcp4** does not verify that assigned address is unused. According to RFC2131, the allocating server should verify that address is no used by sending ICMP echo request.
 - Address renewal (RENEW), rebinding (REBIND), confirmation (CONFIRM), duplication report (DECLINE) and release (RELEASE) are not supported yet.
 - DNS Update is not supported yet.
 - -v (verbose) command line option is currently the default, and cannot be disabled.
-

Chapter 13

DHCPv6 Server

Dynamic Host Configuration Protocol for IPv6 (DHCPv6) is specified in RFC3315. BIND10 provides DHCPv6 server implementation that is described in this chapter. For a description of the DHCPv4 server implementation, see Chapter 12.

The DHCPv6 server component is currently under intense development. You may want to check out [BIND10 DHCP \(Kea\) wiki](#) and recent posts on [BIND10 developers mailing list](#).

The DHCPv4 and DHCPv6 components in BIND10 architecture are internally code named 'Kea'.

Note

As of December 2011, both DHCPv4 and DHCPv6 components are skeleton servers. That means that while they are capable of performing DHCP configuration, they are not fully functional yet. In particular, neither has functional lease databases. This means that they will assign the same, fixed, hardcoded addresses to any client that will ask. See Section 12.4 and Section 13.4 for detailed description.

13.1 DHCPv6 Server Usage

BIND10 provides the DHCPv6 server component since September 2011. It is a skeleton server and can be described as an early prototype that is not fully functional yet. It is mature enough to conduct first tests in lab environment, but it has significant limitations. See Section 13.4 for details.

The DHCPv6 server is implemented as **b10-dhcp6** daemon. As it is not configurable yet, it is fully autonomous, that is it does not interact with **b10-cfgmgr**. To start DHCPv6 server, simply input:

```
#cd src/bin/dhcp6
#./b10-dhcp6
```

Depending on your installation, **b10-dhcp6** binary may reside in `src/bin/dhcp6` in your source code directory, in `/usr/local/bin/b10-dhcp6` or other directory you specified during compilation. At start, server will detect available network interfaces and will attempt to open UDP sockets on all interfaces that are up, running, are not loopback, are multicast-capable, and have IPv6 address assigned. The server will then listen to incoming traffic. Currently supported client messages are SOLICIT and REQUEST. The server will respond to them with ADVERTISE and REPLY, respectively. Since the DHCPv6 server opens privileged ports, it requires root access. Make sure you run this daemon as root.

Note

Integration with **bind10** is planned. Ultimately, **b10-dhcp6** will not be started directly, but rather via **bind10**. Please be aware of this planned change.

13.2 DHCPv6 Server Configuration

The DHCPv6 server does not have lease database implemented yet or any support for configuration, so every time the same set of configuration options (including the same fixed address) will be assigned every time.

At this stage of development, the only way to alter server configuration is to tweak its source code. To do so, please edit `src/bin/dhcp6/dhcp6_srv.cc` file and modify following parameters and recompile:

```
const std::string HARDCODED_LEASE = "2001:db8:1::1234:abcd";
const uint32_t HARDCODED_T1 = 1500; // in seconds
const uint32_t HARDCODED_T2 = 2600; // in seconds
const uint32_t HARDCODED_PREFERRED_LIFETIME = 3600; // in seconds
const uint32_t HARDCODED_VALID_LIFETIME = 7200; // in seconds
const std::string HARDCODED_DNS_SERVER = "2001:db8:1::1";
```

Lease database and configuration support is planned for 2012.

13.3 Supported DHCPv6 Standards

The following standards and draft standards are currently supported:

- RFC3315: Supported messages are SOLICIT, ADVERTISE, REQUEST, and REPLY. Supported options are SERVER_ID, CLIENT_ID, IA_NA, and IAADDRESS.
- RFC3646: Supported option is DNS_SERVERS.

13.4 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.

- Relayed traffic is not supported.
- **b10-dhcp6** provides a single, fixed, hardcoded lease to any client that asks. There is no lease manager implemented. If two clients request addresses, they will both get the same fixed address.
- **b10-dhcp6** does not support any configuration mechanisms yet. The whole configuration is currently hardcoded. The only way to tweak configuration is to directly modify source code. See see Section 13.2 for details.
- Upon start, the server will open sockets on all interfaces that are not loopback, are up, running and are multicast capable and have IPv6 address. Support for multiple interfaces is not coded in reception routines yet, so if you are running this code on a machine that has many interfaces and **b10-dhcp6** happens to listen on wrong interface, the easiest way to work around this problem is to turn down other interfaces. This limitation will be fixed shortly.
- ORO (Option Request Option, a list of options requested by a client) is currently ignored and server assigns DNS SERVER option.
- Temporary addresses are not supported yet.
- Prefix delegation is not supported yet.
- Address renewal (RENEW), rebinding (REBIND), confirmation (CONFIRM), duplication report (DECLINE) and release (RELEASE) are not supported yet.
- DNS Update is not supported yet.
- Interface detection is currently working on Linux only. See Section 14.1 for details.
- `-v` (verbose) command line option is currently the default, and cannot be disabled.

Chapter 14

libdhcp++ library

libdhcp++ is a common library written in C++ that handles many DHCP-related tasks, like DHCPv4 and DHCPv6 packets parsing, manipulation and assembly, option parsing, manipulation and assembly, network interface detection and socket operations, like socket creations, data transmission and reception and socket closing.

While this library is currently used by **b10-dhcp4** and **b10-dhcp6** only, it is designed to be portable, universal library useful for any kind of DHCP-related software.

14.1 Interface detection

Both DHCPv4 and DHCPv6 components share network interface detection routines. Interface detection is currently only supported on Linux systems.

For non-Linux systems, there is currently stub implementation provided. As DHCP servers need to know available addresses, there is a simple mechanism implemented to provide that information. User is expected to create `interfaces.txt` file. Format of this file is simple. It contains list of interfaces along with available address on each interface. This mechanism is temporary and is going to be removed as soon as interface detection becomes available on non-Linux systems. Here is an example of the `interfaces.txt` file:

```
# For DHCPv6, please specify link-local address (starts with fe80::)
# If in doubt, check output of 'ifconfig -a' command.
eth0 fe80::21e:8cff:fe9b:7349

# For DHCPv4, please use following format:
#eth0 192.0.2.5
```

14.2 DHCPv4/DHCPv6 packet handling

TODO: Describe packet handling here, with pointers to wiki

Chapter 15

Statistics

The **b10-stats** process is started by **bind10**. It periodically collects statistics data from various modules and aggregates it.

This stats daemon provides commands to identify if it is running, show specified or all statistics data, show specified or all statistics data schema, and set specified statistics data. For example, using **bindctl**:

```
> Stats show
{
  "Auth": {
    "opcode.iquery": 0,
    "opcode.notify": 10,
    "opcode.query": 869617,
    ...
    "queries.tcp": 1749,
    "queries.udp": 867868
  },
  "Boss": {
    "boot_time": "2011-01-20T16:59:03Z"
  },
  "Stats": {
    "boot_time": "2011-01-20T16:59:05Z",
    "last_update_time": "2011-01-20T17:04:05Z",
    "lname": "4d3869d9_a@jreed.example.net",
    "report_time": "2011-01-20T17:04:06Z",
    "timestamp": 1295543046.823504
  }
}
```

Chapter 16

Logging

16.1 Logging configuration

The logging system in BIND 10 is configured through the Logging module. All BIND 10 modules will look at the configuration in Logging to see what should be logged and to where.

16.1.1 Loggers

Within BIND 10, a message is logged through a component called a "logger". Different parts of BIND 10 log messages through different loggers, and each logger can be configured independently of one another.

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

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

16.1.1.1 `name` (string)

Each logger in the system has a name, the name being that of the component using it to log messages. For instance, if you want to configure logging for the resolver module, you add an entry for a logger named 'Resolver'. This configuration will then be used by the loggers in the Resolver module, and all the libraries used by it.

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

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

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

If there are multiple logger specifications in the configuration that might match a particular logger, the specification with the more specific logger name takes precedence. For example, if there are entries for both '*' and 'Resolver', the resolver module — and all libraries it uses — will log messages according to the configuration in the second entry ('Resolver'). All other modules will use the configuration of the first entry (*). If there was also a configuration entry for 'Resolver.cache', the cache library within the resolver would use that in preference to the entry for 'Resolver'.

One final note about the naming. When specifying the module name within a logger, use the name of the module as specified in **bindctl**, e.g. 'Resolver' for the resolver module, 'Xfrout' for the xfrout module, etc. When the message is logged, the message will include the name of the logger generating the message, but with the module name replaced by the name of the process implementing the module (so for example, a message generated by the 'Auth.cache' logger will appear in the output with a logger name of 'b10-auth.cache').

16.1.1.2 severity (string)

This specifies the category of messages logged. Each message is logged with an associated severity which may be one of the following (in descending order of severity):

- FATAL
- ERROR
- WARN
- INFO
- DEBUG

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

16.1.1.3 output_options (list)

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

The other options for a logger are:

16.1.1.4 debuglevel (integer)

When a logger's severity is set to DEBUG, this value specifies what debug messages should be printed. It ranges from 0 (least verbose) to 99 (most verbose).

If severity for the logger is not DEBUG, this value is ignored.

16.1.1.5 additive (true or false)

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

16.1.2 Output Options

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

16.1.2.1 destination (string)

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

- console
 - file
 - syslog
-

16.1.2.2 output (string)

Depending on what is set as the output destination, this value is interpreted as follows:

destination is 'console' The value of output must be one of 'stdout' (messages printed to standard output) or 'stderr' (messages printed to standard error).

destination is 'file' The value of output is interpreted as a file name; log messages will be appended to this file.

destination is 'syslog' The value of output is interpreted as the **syslog** facility (e.g. *local0*) that should be used for log messages.

The other options for `output_options` are:

16.1.2.2.1 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.

16.1.2.2.2 maxsize (integer)

Only relevant when destination is file, this is maximum file size of output files in bytes. When the maximum size is reached, the file is renamed and a new file opened. (For example, a ".1" is appended to the name — if a ".1" file exists, it is renamed ".2", etc.)

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

16.1.2.2.3 maxver (integer)

Maximum number of old log files to keep around when rolling the output file. Only relevant when destination is 'file'.

16.1.3 Example session

In this example we want to set the global logging to write to the file `/var/log/my_bind10.log`, at severity `WARN`. We want the authoritative server to log at `DEBUG` with `debuglevel 40`, to a different file (`/tmp/debug_messages`).

Start `bindctl`.

```
["login success "]
> config show Logging
Logging/loggers [] list
```

By default, no specific loggers are configured, in which case the severity defaults to `INFO` and the output is written to `stderr`.

Let's first add a default logger:

```
> config add Logging/loggers
> config show Logging
Logging/loggers/ list (modified)
```

The `loggers` value line changed to indicate that it is no longer an empty list:

```
> config show Logging/loggers
Logging/loggers[0]/name "" string (default)
Logging/loggers[0]/severity "INFO" string (default)
Logging/loggers[0]/debuglevel 0 integer (default)
Logging/loggers[0]/additive false boolean (default)
Logging/loggers[0]/output_options [] list (default)
```

The name is mandatory, so we must set it. We will also change the severity as well. Let's start with the global logger.

```
> config set Logging/loggers[0]/name *
> config set Logging/loggers[0]/severity WARN
> config show Logging/loggers
Logging/loggers[0]/name "*" string (modified)
Logging/loggers[0]/severity "WARN" string (modified)
Logging/loggers[0]/debuglevel 0 integer (default)
Logging/loggers[0]/additive false boolean (default)
Logging/loggers[0]/output_options [] list (default)
```

Of course, we need to specify where we want the log messages to go, so we add an entry for an output option.

```
> config add Logging/loggers[0]/output_options
> config show Logging/loggers[0]/output_options
Logging/loggers[0]/output_options[0]/destination "console" string (default)
Logging/loggers[0]/output_options[0]/output "stdout" string (default)
Logging/loggers[0]/output_options[0]/flush false boolean (default)
Logging/loggers[0]/output_options[0]/maxsize 0 integer (default)
Logging/loggers[0]/output_options[0]/maxver 0 integer (default)
```

These aren't the values we are looking for.

```
> config set Logging/loggers[0]/output_options[0]/destination file
> config set Logging/loggers[0]/output_options[0]/output /var/log/bind10.log
> config set Logging/loggers[0]/output_options[0]/maxsize 204800
> config set Logging/loggers[0]/output_options[0]/maxver 8
```

Which would make the entire configuration for this logger look like:

```
> config show all Logging/loggers
Logging/loggers[0]/name "*" string (modified)
Logging/loggers[0]/severity "WARN" string (modified)
Logging/loggers[0]/debuglevel 0 integer (default)
Logging/loggers[0]/additive false boolean (default)
Logging/loggers[0]/output_options[0]/destination "file" string (modified)
Logging/loggers[0]/output_options[0]/output "/var/log/bind10.log" string (modified)
Logging/loggers[0]/output_options[0]/flush false boolean (default)
Logging/loggers[0]/output_options[0]/maxsize 204800 integer (modified)
Logging/loggers[0]/output_options[0]/maxver 8 integer (modified)
```

That looks OK, so let's commit it before we add the configuration for the authoritative server's logger.

```
> config commit
```

Now that we have set it, and checked each value along the way, adding a second entry is quite similar.

```
> config add Logging/loggers
> config set Logging/loggers[1]/name Auth
> config set Logging/loggers[1]/severity DEBUG
> config set Logging/loggers[1]/debuglevel 40
> config add Logging/loggers[1]/output_options
> config set Logging/loggers[1]/output_options[0]/destination file
> config set Logging/loggers[1]/output_options[0]/output /tmp/auth_debug.log
> config commit
```

And that's it. Once we have found whatever it was we needed the debug messages for, we can simply remove the second logger to let the authoritative server use the same settings as the rest.

```
> config remove Logging/loggers[1]
> config commit
```

And every module will now be using the values from the logger named '*'.

16.2 Logging Message Format

Each message written by BIND 10 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:

```
2011-06-15 13:48:22.034 ERROR [b10-resolver.asiolink]
  ASIODNS_OPENSOCK error 111 opening TCP socket to 127.0.0.1(53)
```

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:

2011-06-15 13:48:22.034 The date and time at which the message was generated.

ERROR The severity of the message.

[b10-resolver.asiolink] The source of the message. This comprises two components: the BIND 10 process generating the message (in this case, **b10-resolver**) and the module within the program from which the message originated (which in the example is the asynchronous I/O link module, **asiolink**).

ASIODNS_OPENSOCK The message identification. Every message in BIND 10 has a unique identification, which can be used as an index into the *BIND 10 Messages Manual* (<http://bind10.isc.org/docs/bind10-messages.html>) from which more information can be obtained.

error 111 opening TCP socket to 127.0.0.1(53) A brief description of the cause of the problem. Within this text, information relating to the condition that caused the message to be logged will be included. In this example, error number 111 (an operating system-specific error number) was encountered when trying to open a TCP connection to port 53 on the local system (address 127.0.0.1). The next step would be to find out the reason for the failure by consulting your system's documentation to identify what error number 111 means.