NSD User Manual

NLnet Labs

Dec 06, 2021

GETTING STARTED

Welcome to the documentation of the NLnet Labs Name Server Daemon (NSD), an authoritative DNS name server. It has been developed for operations in environments where speed, reliability, stability and security are of high importance.

NSD is designed with a pure philosophy that prioritises raw performance. This means that if you serve hundreds of thousands or even millions of queries per second, NSD is the leading implementation in the world. This authoritative DNS name server strives to be a reference implementation for emerging standards of the Internet Engineering Task Force (IETF). NSD is distributed free of charge in open source form under the BSD license. For most platforms, packages are available.

This documentation is an open source project maintained by NLnet Labs. is edited via text files in the reStructured-Text markup language and then compiled into a static website/offline document using the open source Sphinx and ReadTheDocs tools.

We always appreciate your feedback and improvements. You can submit an issue or pull request on the GitHub repository, or post a message on the NSD users mailing list. All the contents are under the permissive Creative Commons Attribution 3.0 (CC-BY 3.0) license, with attribution to NLnet Labs.

CHAPTER

ONE

INSTALLATION

To install your own copy of NSD you have two options: use the version provided by your package manager, or download the source and building it yourself.

Installing via the package manager is the easiest option, and on most systems even trivial. The downside is the distributed version can be outdated for some distributions or not have all the compile-time options included that you want. Building and compiling NSD yourself ensures that you have the latest version and all the compile-time options you desire.

1.1 Introduction

NSD consists of two programs: the zone compiler `zonec` and the name server `nsd` itself. The name server works with an intermediate database prepared by the zone compiler from standard zone files.

For NSD operation this means that zones have to be compiled by `zonec` before NSD can use them. All this can be controlled via `rc.d` (SIGTERM, SIGHUP) or **`nsd-control`**, and uses a simple configuration file `nsd.conf`.

1.2 Installing with a package manager

Most package managers maintain a version of NSD, although this version can be outdated if this package has not been updated recently. If you like to upgrade to the latest version, we recommend compiling NSD yourself.

1.2.1 Debian/Ubuntu

Installing NSD with the built-in package manager should be as easy as:

```
sudo apt update
sudo apt install nsd
```

This gives you a compiled and running version of NSD ready to *be configured*.

1.3 Building from source/Compiling

1.3.1 Debian/Ubuntu

First of all, we need our copy of the NSD code. On our website you can find the latest version and the changelog. In this example we'll use version 4.3.7.

```
wget https://nlnetlabs.nl/downloads/nsd/nsd-4.3.7.tar.gz
tar xzf nsd-4.3.7.tar.gz
```

We'll need some tools, such as a compiler and the **make** program.

```
sudo apt update
sudo apt install -y build-essential
```

The library components NSD needs are: `libssl` `libevent`, of which we need the "dev" version.

```
sudo apt install -y libssl-dev
sudo apt install -y libevent-dev
```

We'll also need the tools to build the actual program. For this, NSD uses **make** and internally it uses `flex` and `yacc`, which we need to download as well.

```
sudo apt-get install -y bison
sudo apt-get install -y flex
```

With all the requirements met, we can now start the compilation process in the NSD directory. The first step here is configuring. With `./configure -h` you can look at the extensive list of configurables for NSD. A nice feature is that **configure** will tell you what it's missing during configuration.

```
./configure
```

If **configure** gives no errors, we can continue to actually try compiling NSD using **make**; compilation might take a while.

```
make
```

After successfully compiling, we can install NSD to make it available for the machine.

```
sudo make install
```

We now have fully compiled and installed version of NSD, and can continue to testing it.

1.4 Testing

A simple test to determine if the installation was successful is to invoke the **nsd** command with the `-V` option, which is the "version" option. This shows the version and build options used and proves installation was successful.

```
nsd -v
```

If all the previous steps were successful we can continue to configuring our NSD instance.

Another handy trick you can use during testing is to run NSD in the foreground using the `-d` option and increase the verbosity level using the `-V 3` option. This allows you to see steps NSD takes and also where it fails.

Now that NSD is installed we can *continue to configuring it*.

CHAPTER

TWO

CONFIGURATION

NSD has a vast array of configuration options for advanced use cases. To configure the application, a `nsd.conf` configuration file used. The file format has attributes and values, and some attributes have attributes inside them.

2.1 Configuration file

The configuration is specified in the configuration file, which can be supplied to NSD using the `-c` option. In **`our refence<https://www.nlnetlabs.nl/documentation/nsd/nsd.conf/>`_** (and on your system) an example `nsd.conf` can be found.

The basic principles are:

- The used notation is `attribute: value`
- Comments start with # and extend to the end of a line
- Empty lines are ignored, as is whitespace at the beginning of a line
- Quotes can be used, for names containing spaces, e.g. `"file name.zone"`

The example configuration below specifies options for the NSD server, zone files, primaries and secondaries.

Here is an example config for `example.com`:

```
# Example.com nsd.conf file
# This is a comment.

server:
    server-count: 1 # use this number of cpu cores
    database: ""  # or use "/var/db/nsd/nsd.db"
    zonelistfile: "/var/db/nsd/zone.list"
    username: nsd # the user that will run NSD, can also be "" if user privilige␣
↪protection is not needed
    logfile: "/var/log/nsd.log" # file where all the log messages go
    pidfile: "/var/run/nsd.pid" # use this pid file instead of the platform specific␣
↪default
    xfrdfile: "/var/db/nsd/xfrd.state"

zone:
    name: example.com
    zonefile: /etc/nsd/example.com.zone

zone:
```

(continues on next page)

(continued from previous page)

```
    # this server is master, 192.0.2.1 is the secondary.
    name: masterzone.com
    zonefile: /etc/nsd/masterzone.com.zone
    notify: 192.0.2.1 NOKEY
    provide-xfr: 192.0.2.1 NOKEY

zone:
    # this server is secondary, 192.0.2.2 is master.
    name: secondzone.com
    zonefile: /etc/nsd/secondzone.com.zone
    allow-notify: 192.0.2.2 NOKEY
    request-xfr: 192.0.2.2 NOKEY
```

we provide a sample configuration to get started.

The server settings start with a line with the keyword `server:`. In the server settings set `database: <file>` with the filename of the name database that NSD will use. Set `chroot: <dir>` to run NSD in a chroot-jail. Make sure the zone files, database file, xfrdfile, difffile and pidfile can be accessed from the chroot-jail. Set `username: <user>` to an unprivileged user, for security.

For example:

```
# This is a sample configuration
server:
  database: "/etc/nsd/nsd.db"
  pidfile: "/etc/nsd/nsd.pid"
  chroot: "/etc/nsd/"
  username: nsd
```

After the global server settings to need to make entries for the zones that you wish to serve. For each zone you need to list the zone name, the file name with the zone contents, and access control lists.

```
zone:
  name:       "example.com"
  zonefile: "example.com.zone"
```

The zone file needs to be filled with the correct zone information for primary zones. For secondary zones an empty file will suffice, a zone transfer will be initiated to obtain the secondary zone contents.

Access control lists are needed for zone transfer and notifications.

For a secondary zone list the masters, by IP address. Below is an example of a secondary zone with two primary servers. If a primary only supports AXFR transfers and not IXFR transfers (like NSD), specify the primary as `request-xfr: AXFR <ip_address> <key>`. By default, all zone transfer requests are made over TCP. If you want the IXFR request be transmitted over UDP, use `request-xfr: UDP <ip address> <key>`.

```
zone:
  name: "example.com"
  zonefile: "example.com.zone"
  allow-notify: 168.192.185.33 NOKEY
  request-xfr: 168.192.185.33 NOKEY
  allow-notify: 168.192.199.2 NOKEY
  request-xfr: 168.192.199.2 NOKEY
```

By default, a secondary will fallback to AXFR requests if the primary told us it does not support IXFR. You can

configure the secondary not to do AXFR fallback with:

```
allow-axfr-fallback: "no"
```

For a primary zone, list the secondary servers, by IP address or subnet. Below is an example of a primary zone with two secondary servers:

```
zone:
  name: "example.com"
  zonefile: "example.com.zone"
  notify: 168.192.133.75 NOKEY
  provide-xfr: 168.192.133.75 NOKEY
  notify: 168.192.5.44 NOKEY
  provide-xfr: 168.192.5.44 NOKEY
```

You also can set the outgoing interface for notifies and zone transfer requests to satisfy access control lists at the other end:

```
outgoing-interface: 168.192.5.69
```

By default, NSD will retry a notify up to five times. You can override that value with:

```
notify-retry: 5
```

Zone transfers can be secured with TSIG keys, replace NOKEY with the name of the TSIG key to use. See *Using TSIG* for details.

Since NSD is written to be run on the root name servers, the config file can to contain something like:

```
zone:
  name: "."
  zonefile: "root.zone"
  provide-xfr: 0.0.0.0/0 NOKEY # allow axfr for everyone.
  provide-xfr: ::0/0 NOKEY
```

You should only do that if you're intending to run a root server, NSD is not suited for running a `.` cache. Therefore if you choose to serve the `.` zone you have to make sure that the complete root zone is timely and fully updated.

To prevent misconfiguration, NSD configure has the `--enable-root-server` option, that is by default disabled.

In the config file, you can use patterns. A pattern can have the same configuration statements that a zone can have. And then you can `include-pattern: <name-of-pattern>` in a zone (or in another pattern) to apply those settings. This can be used to organise the settings.

The **`nsd-control`** tool is also controlled from the `nsd.conf` config file. It uses TLS encrypted transport to 127.0.0.1, and if you want to use it you have to setup the keys and also edit the config file. You can leave the remote-control disabled (the secure default), or opt to turn it on:

```
# generate keys
nsd-control-setup
```

```
# edit nsd.conf to add this
remote-control:
  control-enable: yes
```

By default **`nsd-control`** is limited to localhost, as well as encrypted, but some people may want to remotely administer their nameserver. What you then do is setup **`nsd-control`** to listen to the public IP address, with

`control-interface:` <IP> after the control-enable statement.

Furthermore, you copy the key files `/etc/nsd/nsd_server.pem` `/etc/nsd/nsd_control.*` to a remote host on the internet; on that host you can run **nsd-control** with `-c` which references same IP address `control-interface` and references the copies of the key files with `server-cert-file`, `control-key-file` and `control-cert-file` config lines after the `control-enable` statement. The nsd-server authenticates the nsd-control client, and also the **nsd-control** client authenticates the nsd-server.

When you are done with the configuration file, check the syntax using

```
nsd-checkconf <name of configfile>
```

The zone files are read by the daemon, which builds `nsd.db` with their contents. You can start the daemon with:

```
nsd
or with "nsd-control start" (which execs nsd again).
or with nsd -c <name of configfile>
```

To check if the daemon is running look with **ps**, **top**, or if you enabled command:*nsd-control*:

```
nsd-control status
```

To reload changed zone files after you edited them, without stopping the daemon, use this to check if files are modified:

```
kill -HUP `cat <name of nsd pidfile>`
```

If you enabled **nsd-control**, you can re-read with:

```
nsd-control reload
```

With **nsd-control** you can also reread the config file, in case of new zones, etc.

```
nsd-control reconfig
```

To restart the daemon:

```
/etc/rc.d/nsd restart  # or your system(d) equivalent
```

To shut it down (for example on the system shutdown) do:

```
kill -TERM <pid of nsd>
or nsd-control stop
```

NSD will automatically keep track of secondary zones and update them when needed. When primary zones are updated and reloaded notifications are sent to secondary servers.

The zone transfers are applied to `nsd.db` by the daemon. To write changed contents of the zone files for secondary zones to disk in the text-based zone file format, issue **nsd-control** write.

NSD will send notifications to secondary zones if a primary zone is updated. NSD will check for updates at primary servers periodically and transfer the updated zone by AXFR/IXFR and reload the new zone contents.

If you wish exert manual control use **nsd-control notify**, **transfer** and **force_transfer** commands. The transfer command will check for new versions of the secondary zones hosted by this NSD. The notify command will send notifications to the secondary servers configured in `notify:` statements.

CHAPTER

THREE

LOGGING

NSD does not provide any DNS logging. We believe that this is a separate task and has to be done independently from the core operation. This consciously is not part of NSD itself in order to keep NSD focused and minimise its complexity. It is better to leave logging and tracing to separate dedicated tools.

The CAIDA dnsstat tool can easily be configured and/or modified to suit local statistics requirements without any danger of affecting the name server itself. We have run `dnsstat` on the same machine as NSD, and we would recommend using a multiprocessor if performance is an issue. Of course, `dnsstat` can also run on a separate machine that has MAC layer access to the network of the server.

The **`nsd-control`** tool can output some statistics, with **`nsd-control stats`** and **`nsd-control stats_noreset`**. In contrib/nsd_munin_ there is a Munin grapher plugin that uses it. The output of **`nsd-control stats`** is easy to read (text only) with scripts. The output values are documented on the **`nsd-control`** man page.

Another available tool is dnstop, which displays DNS statistics on your network.

CHAPTER

FOUR

USING TSIG (TRANSACTION SIGNATURE)

NSD supports TSIG (Transaction Signature) for zone transfer and for notify sending and receiving, for any query to the server.

TSIG keys are based on shared secrets. These must be configured in the config file. To keep the secret in a separate file use `include: "filename"` to include that file.

An example TSIG key named `sec1_key`.

```
key:
  name: "sec1_key"
  algorithm: hmac-md5
  secret: "6KM6qiKfwfEpamEq72HQdA=="
```

This key can then be used for any query to the NSD server. NSD will check if the signature is valid, and if so, return a signed answer. Unsigned queries will be given unsigned replies.

The key can be used to restrict the access control lists, for example to only allow zone transfer with the key, by listing the key name on the access control line.

```
# provides AXFR to the subnet when TSIG is used.
provide-xfr: 10.11.12.0/24 sec1_key
# allow only notifications that are signed
allow-notify: 192.168.0.0/16 sec1_key
```

If the TSIG key name is used in `notify` or `request-xfr` lines, the key is used to sign the request/notification messages.

CHAPTER

FIVE

ZONE EXPIRY OF SECONDARY ZONES

NSD will keep track of the status of secondary zones, according to the timing values in the SOA record for the zone. When the refresh time of a zone is reached, the serial number is checked and a zone transfer is started if the zone has changed. Each primary server is tried in turn.

Master zones cannot expire so they are always served. Zones are interpreted primary zones if they have no `request-xfr:` statements in the config file.

After the expire timeout (from the SOA record at the zone apex) is reached, the zone becomes expired. NSD will return `SERVFAIL` for expired zones, and will attempt to perform a zone transfer from any of the primaries. After a zone transfer succeeds, or if the primary indicates that the SOA serial number is still the same, the zone will be OK again.

In contrast with e.g. BIND, the inception time for a secondary zone is stored on disk (in `xfrdfile: "xfrd.state"`), together with timeouts. If a secondary zone acquisition time is recent enough, this means that NSD can start serving a zone immediately on loading, without querying the primary server.

If your secondary zone has expired and no primaries can be reached, but you still want NSD to serve the zone, then you can delete the `xfrd.state` file, but leave the zone file for the zone intact. Make sure to stop NSD before you delete the file, as NSD writes it on exit. Upon loading NSD will treat the zone file that you as operator have provided as recent and will serve the zone. Even though NSD will start to serve the zone immediately, the zone will expire after the timeout is reached again. NSD will also attempt to confirm that you have provided the correct data by polling the primaries. So when the primary servers come back up, it will transfer the updated zone within <retry timeout from SOA> seconds.

In general it is possible to provide zone files for both primary and secondary zones manually (say from email or rsync). Reload with SIGHUP or **`nsd-control reload`** to read the new zone file contents into the name database. When this is done the new zone will be served. For primary zones, NSD will issue notifications to all configured `notify:` targets. For secondary zones the above happens; NSD attempts to validate the zone from the primary (checking its SOA serial number).

CHAPTER

SIX

INTERFACES

NSD will by default bind itself to the system default interface and service IPv4 and if available also IPv6. It is possible to service only IPv4 or IPv6 using the `-4`, `-6` command line options, or the `ip4-only` and `ip6-only` config file options.

The command line option `-a` and config file option ip-address can be given to bind to specific interfaces. Multiple interfaces can be specified, which is useful for two reasons:

- The specific interface bound will result in the OS bypassing routing tables for the interface selection. This results in a small performance gain. It is not the performance gain that is the problem: sometimes the routing tables can give the wrong answer, see the next point.
- The answer will be routed via the interface the query came from. This makes sure that the return address on the DNS replies is the same as the query was sent to. Many resolvers require the source address of the replies to be correct. The `ip-address:` option is easier than configuring the OS routing table to return the DNS replies via the correct interface.

The above means that even for systems with multiple interfaces where you intend to provide DNS service to all interfaces, it is prudent to specify all the interfaces as `ip-address` config file options.

With the config file option `ip-transparent` you can allow NSD to bind to non-local addresses.

CHAPTER

SEVEN

TUNING

In version 4.3.0 of NSD, additional functionality was added to increase performance even more. Most notably, this includes processor affinity.

NSD is performant by design because it matters when operators serve hundreds of thousands or even millions of queries per second. We strive to make the right choices by default, like enabling the use of `libevent` at the configure stage to ensure the most efficient event mechanism is used on a given platform. e.g. `epoll` on Linux and `kqueue` on FreeBSD. Switches are available for operators who know the implementation on their system behaves correctly, like enabling the use of `recvmmsg` at the configure stage (`--enable-recvmmsg`) to read multiple messages from a socket in one system call.

By default NSD forks (only) one server. Modern computer systems however, may have more than one processor, and usually have more than one core per processor. The easiest way to scale up performance is to simply fork more servers by configuring server-count: to match the number of cores available in the system so that more queries can be answered simultaneously. If the operating system supports it, ensure `reuseport:` is set to `yes` to distribute incoming packets evenly across server processes to balance the load.

A couple of other options that the operator may want to consider:

1. Memory usage can be lowered (around 50%) by using zone files and disable the on-disk database by setting `database: ""`.
2. TCP capacity can be significantly increased by setting `tcp-count: 1000` and `tcp-timeout: 3`. Set `tcp-reject-overflow: yes` to prevent the kernel connection queue from growing.

7.1 Processor Affinity

The aforementioned settings provide an easy way to increase performance without the need for in-depth knowledge of the hardware. For operators that require even more throughput `cpu-affinity` is available.

The operating system's scheduling-algorithm determines which core a given task is allocated to. Processors build up state — e.g. by keeping frequently accessed data in cache memory — for the task that it is currently executing. Whenever a task switches cores, performance is degraded because the core it switched to has yet to build up said state. While this scheduling-algorithm works just fine for general-purpose computing, operators may want to designate a set of cores for best performance. The `cpu-affinity` family of configuration options was added to NSD specifically for that purpose.

Processor affinity is currently supported on Linux and FreeBSD. Other operating systems may be supported in the future, but not all operating systems that can run NSD support CPU pinning. To fully benefit from this feature, one must first determine which cores should be allocated to NSD. This requires some knowledge of the underlying hardware, but generally speaking every process should run on a dedicated core and the use of Hyper-Threading cores should be avoided to prevent resource contention. List every core designated to NSD in `cpu-affinity` and bind each server process to a specific core using `server-<N>-cpu-affinity` and `xfrd-cpu-affinity` to improve L1/L2 cache hit rates and reduce pipeline stalls/flushes.

```
server:
  server-count: 2
  cpu-affinity: 0 1 2
  server-1-cpu-affinity: 0
  server-2-cpu-affinity: 1
  xfrd-cpu-affinity: 2
```

7.2 Partition Sockets

`ip-address:` options in the `server:` clause can be configured per server or set of servers. Sockets configured for a specific server are closed by other servers on startup. This improves performance if a large number of sockets are scanned using `select/poll` and avoids waking up multiple servers when a packet comes in, known as the thundering herd problem. Though both problems are solved using a modern kernel and a modern I/O event mechanism, there is one other reason to partition sockets, explained below.

```
server:
  ip-address: 192.0.2.1 servers=1
```

7.3 Bind to Device

`ip-address:` options in the server: clause can now also be configured to bind directly to the network interface device on Linux (`bindtodevice=yes`) and to use a specific routing table on FreeBSD (`setfib=<N>`). These were added to ensure UDP responses go out over the same interface the query came in on if there are multiple interfaces configured on the same subnet, but there may be some performance benefits as well as the kernel does not have to go through the network interface selection process.

```
server:
  ip-address: 192.0.2.1 bindtodevice=yes setfib=<N>
```

Note: FreeBSD does not create extra routing tables on demand. Consult the FreeBSD Handbook, forums, etc. for information on how to configure multiple routing tables.

7.4 Combining Options

Field tests have shown best performance is achieved by combining the aforementioned options so that each network interface is essentially bound to a specific core. To do so, use one IP address per server process, pin that process to a designated core and bind directly to the network interface device.

```
server:
  server-count: 2
  cpu-affinity: 0 1 2
  server-1-cpu-affinity: 0
  server-2-cpu-affinity: 1
  xfrd-cpu-affinity: 2
  ip-address: 192.0.2.1 servers=1 bindtodevice=yes setfib=1
  ip-address: 192.0.2.2 servers=2 bindtodevice=yes setfib=2
```

The above snippet serves as an example on how to use the configuration options. Which cores, IP addresses and routing tables are best used depends entirely on the hardware and network layout. Be sure to test extensively before using the options.

CHAPTER

EIGHT

MANUAL PAGES

You can find the manual pages of the latest version of NSD here, rendered in HTML format:

- nsd(8), the daemon
- nsd.conf(5), the extensive configuration file reference
- nsd-checkconf(8), the configuration checker
- nsd-checkzone(8), the zone checker
- nsd-control(8), the nsd control program

CHAPTER

NINE

CONFIGURE OPTIONS

NSD can be configured using GNU autoconf's configure script. In addition to standard configure options, one may use the following:

CC=compiler Specify the C compiler. The default is gcc or cc. The compiler must support ANSI C89.

CPPFLAGS=flags Specify the C preprocessor flags. Such as `-I<includedir>`.

CFLAGS=flags Specify the C compiler flags. These include code generation, optimisation, warning, and debugging flags. These flags are also passed to the linker.

The default for gcc is `-g -O2`.

LD=linker Specify the linker (defaults to the C compiler).

LDFLAGS=flags Specify linker flags.

LIBS=libs Specify additional libraries to link with.

--enable-root-server Configure NSD as a root server. Unless this option is specified, NSD will refuse to serve the `.` zone as a misconfiguration safeguard.

--disable-ipv6 Disables IPv6 support in NSD.

--enable-checking Enable some internal development checks. Useful if you want to modify NSD. This option enables the standard C "assert" macro and compiler warnings.

This will instruct NSD to be stricter when validating its input. This could lead to a reduced service level.

--enable-bind8-stats Enables BIND8-like statistics.

--enable-ratelimit Enables rate limiting, based on query name, type and source.

--enable-draft-rrtypes Enables draft RRtypes.

--with-configdir=dir Specified, NSD configuration directory, default `/etc/nsd`.

--with-nsd_conf_file=path Pathname to the NSD configuration file, default `/etc/nsd/nsd.conf`.

--with-pidfile=path Pathname to the NSD pidfile, default is platform specific, mostly `/var/run/nsd.pid`.

--with-dbfile=path Pathname to the NSD database, default is `/etc/nsd/nsd.db`.

--with-zonesdir=dir NSD default location for master zone files, default `/etc/nsd/`.

--with-user=username User name or ID to answer the queries with, default is `nsd`.

--with-facility=facility Specify the syslog facility to use. The default is LOG_DAEMON. See the syslog(3) manual page for the available facilities.

--with-libevent=path Specity the location of the `libevent` library (or libev). `--with-libevent=no` uses a builtin portable implementation (select()).

--with-ssl=path Specify the location of the OpenSSL libraries. OpenSSL 0.9.7 or higher is required for TSIG support.

--with-start_priority=number Startup priority for NSD.

--with-kill_priority=number Shutdown priority for NSD.

--with-tcp-timeout=number Set the default TCP timeout (in seconds). The default is 120 seconds.

--disable-nsec3 Disable NSEC3 support. With NSEC3 support enabled, very large zones, also non-NSEC3 zones, use about 20% more memory.

--disable-minimal-responses Disable minimal responses. If disabled, responses are more likely to get truncated, resulting in TCP fallback. When enabled (by default) NSD will leave out RRsets to make responses fit inside one datagram, but for shorter responses the full normal response is carried.

--disable-largefile Disable large file support (64 bit file lengths). Makes off_t a 32bit length during compilation.

CHAPTER

TEN

DIAGNOSING NSD LOG ENTRIES

NSD will print log messages to the system log (or `logfile:` configuration entry). Some of these messages are covered here.

Reload process <pid> failed with status <s>, continuing with old database This log message indicates the reload process of NSD has failed for some reason. This can be anything from a missing database file to internal errors.

snipping off trailing partial part of <ixfr.db> The file `ixfr.db` contains only part of expected data. The corruption is removed by snipping off the trailing part.

memory recyclebin holds <num> bytes This is printed for every reload. NSD allocates and deallocates memory to service IXFR updates. The recycle bin holds deallocated memory ready for future use. If the number grows too large, a restart resets it.

xfrd: max number of tcp connections (32) reached This line is printed when more than 32 zones need a zone transfer at the same time. The value is a compile constant (`xfrd-tcp.h`), but if this happens often for you, we could make this a config option. NSD will reuse existing TCP connections to the same primary (determined by IP address) to transfer up to 64k zones from that primary. Thus this error should only happen with more than 32 primaries or more than 64*32=2M zones that need to be updated at the same time.

If this happens, more zones have to wait until a zone transfer completes (or is aborted) before they can have a zone transfer too. This waiting list has no size limit.

error: <zone> NSEC3PARAM entry <num> has unknown hash algo <number> This error means that the zone has NSEC3 chain(s) with hash algorithms that are not supported by this version of NSD, and thus cannot be served by NSD. If there are also no NSECs or NSEC3 chain(s) with known hash algorithms, NSD will not be able to serve DNSSEC authenticated denials for the zone.

CHAPTER

ELEVEN

GRAMMAR FOR DNS ZONE FILES

Note: It is near impossible to write a clean lexer/grammar for DNS (**RFC 1035**) zone files. At first it looks like it is easy to make such a beast, but when you start implementing it the details make it messy.

Since as early as NSD 1.4, the parser relies on Bison and Flex, tools for building programs that handle structured input. Compared to the previous implementation there is a slight decrease in speed (10-20%), but as the zone compiler is not critical to the performance of NSD, this not too relevant. The lexer part is located in the file zlexer.lex, the grammar is in zparser.y.

11.1 Zone File Lexer

Finding a good grammar and lexer for BIND zone files is rather hard. There are no real keywords and the meaning of most of the strings depends on the position relative to the other strings. An example, the following is a valid SOA record:

```
$ORIGIN example.org.
    SOA    soa    soa    ( 1 2 3 4 5 6 )
```

This SOA records means the administrator has an email address of `soa@example.org`. and the first nameserver is named `soa.example.org`. Both completely valid. The numbers are of course totally bogus.

Another example would be:

```
$ORIGIN example.org.
    SOA    soa    soa    ( 1 2 ) ( 3 4 ) ( 5 ) ( 6 )
```

The parsing of parentheses was also not trivial. Whitespace is also significant in zonefiles. The TAB before SOA has to be returned as previous_domain token by the lexer. Newlines inside parentheses are returned as SPACE which works but required some changes in the definitions of the resource records.

As shown above a simple `grep -i` for SOA does not do the trick. The lexer takes care of this tricky part by using an extra variable `in_rr` which is an enum containing: `outside`, `expecting_dname`, `after_dname`, `reading_type`. The semantics are as follows:

- `outside`, not in an RR (start of a line or a $-directive);
- `expecting_dname`, parse owner name of RR;
- `after_dname`, parse ttl, class;
- `reading_type`, we expect the RR type now;

With `in_rr` the lexer can say that in the first example above the first SOA is the actual record type, because it is located after a TAB. After we have found the TAB we set `in_rr` to `after_dname` which means we actually are expecting a RR type.

Again this is also not trivial because the class (IN) and TTL are also optional, if there are not specified we should substitute the current defaults from the zone we are parsing (this happens in the grammar). A DNS zone file is further complicated by the unknown RR record types.

11.2 Zone File Grammar

After the lexer was written the grammar itself is quite clean and nice. The basic idea is that every RR consists of single line (the parentheses are handled in the lexer - so this really is the case). If a line is not a RR it is either a comment, empty or a $-directive. Some $-directives are handled inside the lexer ($INCLUDE) while others ($ORIGIN) must be dealt with inside the grammer.

An RR is defined as:

```
rr:     ORIGIN SP rrrest
```

and:

```
rrrset: classttl rtype
```

And then we have a whole list of:

```
rtype: TXT sp rdata_txt
       | DS sp rdata_ds
       | AAAA sp rdata_aaaa
```

which are then parsed by using the `rdata_` rule. Shown here is the one for the SOA:

```
rdata_soa:  dname sp dname sp STR sp STR sp STR sp STR sp STR trail
    {
        /* convert the soa data */
        zadd_rdata_domain( current_parser, $1); /* prim. ns */
        zadd_rdata_domain( current_parser, $3); /* email */
        zadd_rdata_wireformat( current_parser,  \
                zparser_conv_rdata_period(zone_region, $5.str) ); /* serial */
        zadd_rdata_wireformat( current_parser,  \
                zparser_conv_rdata_period(zone_region, $7.str) ); /* refresh */
        zadd_rdata_wireformat( current_parser,  \
                zparser_conv_rdata_period(zone_region, $9.str) ); /* retry */
        zadd_rdata_wireformat( current_parser,  \
                zparser_conv_rdata_period(zone_region, $11.str) ); /* expire */
        zadd_rdata_wireformat( current_parser,  \
                zparser_conv_rdata_period(zone_region, $13.str) ); /* minimum */

        /* XXX also store the minium in case of no TTL? */
        if ( (current_parser->minimum = zparser_ttl2int($11.str) ) == -1 )
            current_parser->minimum = DEFAULT_TTL;
    };
```

The semantic actions in the grammer store the RR data for processing by the zone compiler. The resulting database is then used by NSD the serve the data.

INDEX