robdns 0.2 - manual

# Introduction

ROBDNS is the *fastest* DNS server for handling *authoritative* queries, as well as being highly *scriptable* for customization and experimentation. It is designed for *Internet-scale* applications, such as *root* servers, *TLD*s with 100 million names, *hosting* with 1 million zones, and *blackholing* with thousands of updates per second. It’s is designed to be exposed to the public Internet under constant attack by hackers and DDoS.

It is currently in *prototype* stage of development. It has the basic features for a working DNS server in order to benchmark and test features, but requires extensive testing before it can be relied upon for real use.

The server is broadly compatible with BIND9, reading the same *zonefiles* and *conf* files. Configuration parameters often have the same names as in BIND9, and some BIND9 configuration files can be used directly.

The purpose of this server is to act as an Internet-exposed *slave server* (and *secondary master*), protecting a *hidden master* from attack. It is *authoritative only*. It cannot perform recursive queries, so cannot be used as a *resolver* or *proxy* *slave*. It does only a few things well.

Zones can be either traditional slave zones updated through DNS (refresh/NOTIFY, IXR/AXFR, UPDATE), or *pseudo master* zones which read updates from the zonefiles on the drive. Presumably, such zonefiles come from a hidden master through an out-of-band system such as rsync or scripts.

It is a *modern* server, being up-to-date with the latest *RR types*, *EDNS0*, *NSID*, *TSIG*, *RRL*, and so on. It supports *DynDNS* features (NOTIFY, IXFR, UPDATE)*,* including zone creation/deletion for *superslave* mode. It supports DNSSEC record types, but has no ability to sign or validate zones (though, in theory, automatic signing can be scripted).

This software is designed for *cybersecurity*. There is a *bug bounty* for any security vulnerabilities. The software is written according to *langsec* principles. The software goes through *static analysis*, *dynamic analysis*, and *fuzzing* in order to verify robustness. Software patches can be applied with no effectively no server downtime, and bad patches quickly reversed.

The code is *portable* and should run anywhere. It is specifically tested on *Windows*, *Mac OS X*, and *Linux*. It is specifically tested on *x86* and *ARM*, in both *32-bit* and *64-bit* modes. However, the majority of testing is on Linux and 64-bit x86. The code is massively *multicore*.

## Internet scale DNS

The purpose of *ROBDNS* is to explore Internet-scale DNS performance. DNS is an essential part of Internet infrastructure. However, rather than being written like routers and firewalls, DNS software is written like typical user apps. This project changes that, creating a DNS server written with the same infrastructure techniques as hardware appliances.

Specifically, the software separates the *data-plane* (serving DNS queries) from the *control-plane* (IXFRs to the master). The data-plane adapter may be completely separated from the operating-system’s network stack, using such drivers as *netmap*, *PF\_RING*, or *DPDK*. Whereas a typical operating-system struggles receiving a million packets-per-second, bypassing the operating system makes it easy to receive 30 million packets-per-second. Using a data-plane adapter thus allows for dramatically higher performance that would not be possible otherwise.

While a typical DNS server handles fewer than a hundred queries per second, an Internet exposed server can be flooded with millions of queries per second when under DDoS attack. This can be far more than any machine can handle, even with perfect software. Improving software, though, means fewer machines are needed to cope with such attacks. This software is not simply faster than other servers, but 10 to 100 times faster. A single machine running ROBDNS can answer queries at a rate 100 times faster than BIND9. Using this software would thus significantly reduce hardware requirements.

The intended deployment is as a widely *anycasted* service through the Internet in a *hosting* environment with a million customers. Each server should be able to handle a full 10-gbps of incoming queries, and a spread of many anycast machines should diffuse DDoS attacks.

## Scripting system

This software allows for scripts written in the *Lua* programming language.

One use of Lua is *hooking* existing internal events. A Lua script sets a filter to select just the events the script it interested in, and a callback for when the event triggers. An example would be to set a hook for certain incoming queries.

Another use of the Lua would be for *rewriting* zone data. An example would for load balancing, whereby a the ordering of an RRSet would be continually changed depending upon the load of the target servers. Similarly, the order of an RRset might change for geo-tagging purposes.

Another use of Lua would be *live resource records*, which generate a new response for each query. An example use of such a system would be developing a module that provides access to Twitter via TXT records.

Another example would be the live signing of zones with DNSSEC.

## Slaves and pseudo-masters

This server is intended to operate only as an *authoritative slave server*. It is missing features expected of master servers, such as an SQL database backend and automatic DNSSEC signing of zones. But it does have a sort of *pseudo-master* functionality operating from standard format zonefiles.

The standard *slave* zone works in much the same way as in other DNS servers. The server will check for updates every *refresh* interval, or be notified (with *NOTIFY*) of changes by the master. It will then initiate an *IXFR* transaction to download changes from the master, or if that fails, a full *AXFR* to transfer the entire zone. It may also receive *UPDATE* transactions from the master. If a zonefile exists in the cache, it will read that file on startup. Whenever a change is received from a zone, it will journal the changes to the disk, and eventually update the zonefile in the cache. Except at startup, the zonefile in the cache is *write-only*.

A zone marked *master* is still considered a slave, but one where some other process updates the zonefiles. One example would be using *rsync* to copy file updates among servers. Another example would be where the true master isn’t a DNS server at all, but an SQL database and customer portal – where scripts rewrite the zonefiles in response to changes to the SQL database. The server will look for changes to the files, either immediately when the file/directory timestamps changes, when *refresh* expires, or when it receives a *SIGHUP*. The files are *read-only*, the server will never make a change to the files itself.

Both types support *superslave* operation, whereby masters can create or destroy zones dynamically. Both types support dynamic DNS by forwarding *UPDATE* from non-masters to the real master.

# Building and Installing

## Building from source

This is an open-source project, with code hosted on GitHub in the usual manner. It compiles on Windows, Mac OS X, Linux, and most other platforms in the usual manners. It has few dependencies -- all that is needed on Linux is the *build-essential* package, on Mac OS X the developer tools, and on Windows either MinGW or Visual Studio Express.

To get the source code, you can run the *git* command:

$ git clone https://github.com/robertdavidgraham/robdns

Alternatively, you may want to fetch the zip file containing the latest code.

$ wget https://github.com/robertdavidgraham/robdns/archive/master.zip

$ unzip master.zip

You will probably need to download libpcap/winpcap, and then you can simply build it:

$ cd robdns

$ make

$ make test

Always run the regression test suite, *make test*, after a build in order verify that everything is working correctly.

### Building on Debian/Ubuntu Linux

On a raw system, you’ll need the following:

$ apt-get install build-essentials libpcap-dev

$ git clone https://github.com/robertdavidgraham/robdns

$ cd robdns

$ make test

### Building on Mac OS X

To build the software, you’ll need the development tools from Apple. They are free.

After that, there are two options for building the software. The first is just like Linux:

$ git clone https://github.com/robertdavidgraham/robdns

$ cd robdns

$ make test

The second option is to use the XCode project in the subdirectory *xcode4*.

### Building on Windows

To build the software, there are two options. The first is to use Microsoft’s compiler. You can download *Visual Studio Express*, then load the workspace project in the subdirectory *robdns/vs10*.

The second option is GNU compiler in the *msys* collection, in which case the same technique of simply typing “make” as for Linux will work.

For raw socket performance, you’ll also need to install the WinPCap library from [www.winpcap.org](http://www.winpcap.org).

## Running as service

This software runs in the usual manner as a service on Windows, Mac OS X, and Linux. Since the most common deployment will be under Linux, we’ll describe how Linux services work.

## Resource Requirements and Planning

### Hardware requirements

Desktop-class systems and old hardware work fine for 10-gbps operation. Large server iron is likely overkill.

Virtual machine are often limited in the raw network packet rate, so should not be used for more than 1-gbps operation.

### CPU requirements

No special CPUs are needed. Both 32-bit and 64-bit processors work fine. However, for large zones, or many zones, each in millions range, 64-bit processors will be needed to address the memory. For 1-gpbs network connections, low-power 1-ghz processors will work fine. For 10-gbps, larger multicore processors will be needed.

The software scales well with multiple cores. It’s only been tested up to 6 hyperthreaded cores, but it appears to scale well beyond that.

It’s been tested with ARM and x86 processors. It should compile for any processor type, although there may be some issues with the thread synchronization primitives for other processors.

The software does take advantage of modern software features like SSE and AES instructions. This has only a minor effect on performance, though, so older processors without these extensions should work fine.

### Memory requirements

The amount of memory needed is roughly twice the size of the zonefile. Thus, the 8-gigabyte .com zonefile will need about 15-gigabytes of RAM in order to load. A million small zones with 500 byte zonefiles requires 1-gigabytes of memory.

When doing a full software update, two processes will be running side-by-side with the full zonefiles. Therefore, twice the memory will be needed to support this feature.

### Network adapter

Instead of using the sockets API to the operating-system’s TCP/IP stack, the software contains an optional custom TCP/IP stack that runs many times faster using *zero-copy* drivers. These are special drivers that DMA packets directly to/from user-mode software. Three different drivers can be used: *PF\_RING,* *DPDK*, and *netmap*. All of these drivers require network adapters based on Intel chips. For 1-gbps operation (instead of 10-gbps), the *netmap* driver also supports RealTek chips.

### Supported operating systems

The software compiles and runs on most all operation systems. It is specifically tested on recent versions of Windows, Mac OX X, and Linux.

The *zero-copy* driver options are only available for Linux and FreeBSD.

The code is sufficiently portable that it should, in theory, run on any operating system.

### Network infrastructure

The software is designed to withstand DDoS attacks. It’s likely that the network infrastructure resources will fail before the server does. Even as low as 1-gbps, switches and routers start failing under high packet rate. DNS packets are much smaller than typical web traffic, meaning the packet rate is much higher in order to fill 1-gbps. Many network engineers have not tested for these high packet rates, so their infrastructure often fails under the load.

Thus, while this software will run in any network infrastructure, it’s probably not worth running this software (as compared to more feature rich software) when the network infrastructure is not sufficient.

### Dual-homed environment

The system splits communications into *data-plane* and *control-plane*, which may require separate network adapters: one exposed to hackers and DDoS attacks from the Internet, and the second which may take a different, more protected route to the network. Ideally, the data-plane and control-plane connections are widely spaced in the data center, taking different routes to the Internet, so that an attack on the data-plane adapter will not adversely affect control-plane communications.

# Basic Configuration

The server supports roughly the same file-format as BIND9. As the server can only be configured for *authoritative slave* and *secondary master* operation, only those options are supported.

## Quick testing

Instead of a configuration, the software supports easy testing on the command-line, which tries to infer intent. For example, to start running, simply type:

$ **robdns example.zone**

This will parse the indicated zonefile and start listening for queries on port 53, on both TCP and UDP, on both *0.0.0.0* and *[::]*. You can use the *dig* utility to query this zone to verify that it’s working.

The software contains its own TCP/IP stack. To test it, type something like the following:

$ **robdns example.zone eth0 10.1.2.3**

The software will infer that you want to use the *eth0* network adapter, but that instead of using the IP addresses of the operating-system’s TCP/IP stack, you want to *spoof* an unused address. You might then flood it with queries in order to benchmark the difference in speed between the operating-system’s stack and the custom stack.

## Sample configuration

The following is a sample configuration for a typical dynamically updated slave zone.

options {

directory "/var/cache/named";

listen-on port 53 { 192.168.17.254; }

listen-on port 53 { 2001:0db8:100::4; };

};

key MyKey {

algorithm HMAC-MD5;

secret "7zTrFAk8z5YP2IaHNdy0ig=="

};

zone "example.com" {

type slave;

file "example.zone";

allow-update { key MyKey; };

};

This configuration, which works the same way as for BIND9. It reads the zonefile contents from *example.zone* and starts serving queries on that zone. It allows updates from a master using the specified secret key.

The key was generated by generating random data:

$ **dd if=/dev/urandom bs=1 count=16 | base64**

7zTrFAk8z5YP2IaHNdy0ig==

16+0 records in

16+0 records out

16 bytes (16 B) copied, 6.6499e-05 s, 241 kB/s

One way of updating the records is using the *nsupdate* program from *bind-utils*. The following is an example:

$ **nsupdate –y HMAC-MD5:MyKey:7zTrFAk8z5YP2IaHNdy0ig==**

> **update add www.example.com 600 cname www1.example.com.**

> **send**

Standard zonefile format for *example.zone* file would be used, such as having the following contents:

$TTL 86400 ; 24 hours could have been written as 24h or 1d

$ORIGIN example.com.

@ 1D IN SOA ns1.example.com. hostmaster.example.com. (

2012022401 ; serial

3H ; refresh

15 ; retry

1w ; expire

3h ; minimum

)

IN NS ns1.example.com. ; in the domain

IN NS ns2.smokeyjoe.com. ; external to domain

IN MX 10 mail.another.com. ; external mail provider

; server host definitions

ns1 IN A 192.168.0.1 ;name server definition

www IN A 192.168.0.2 ;web server definition

ftp IN CNAME www.example.com. ;ftp server definition

; non server domain hosts

bill IN A 192.168.0.3

fred IN A 192.168.0.4

This example also works on 10-year old versions of BIND9. The major difference is that this server will run many times faster than BIND9.

## Signals

On UNIX, the service will respond as expected to signals. SIGHUP will reload zone information. SIGTERM and SIGINT will terminate the program gracefully.

Note that SIGHUP will only reload zonefile information and change most configuration parameters, but not all parameters. Changing everything requires a full *reload* of the process.

Note that zones of type *slave* only read the zonefile cache on startup. Afterwards, zonefiles in the cache are written with updated information, but never read. A SIGHUP in such cases will not cause a reload of those zones from the cache. Conversely, zones of type *master* will reload any changed zonefiles in response to a SIGHUP – indeed, that’s expected to happen quite often.

For efficiency, as little work as possible is done in response to a SIGHUP. If none of the configuration files have changed, then the configuration will not be reloaded. All zonefiles which have not changed will not be reloaded. The expectation is that that, in a zone *master* configuration, that other processes will be frequently changing one of the zonefiles and sending a SIGHUP to the process.

# Advanced DNS Features

## Notify

DNS *NOTIFY* [RFC1996] is a packet sent to all slaves whenever the master’s copy of the zone has changed.

As a slave, when we receive a NOTIFY from the master, we’ll initiate an IXFR transaction to retrieve the new zone information from the master.

As a secondary master, once we’ve retrieved the latest zone information, we will transmit NOTIFY messages to secondary slaves.

All NOTIFY (and IXFR/AXFR) messages must be signed with TSIG keys.

### Superslave

As defined by the RFCs, only existing zones can be updated with NOTIFY packets. However, we support the PowerDNS *superslave* extension, whereby NOTIFY packets from a trusted master can cause new zones to be created. These new zones inherit the default zone configuration for that master.

## DynDNS UPDATE

DNS *UPDATE* [RFC2136] is a packet sent to update the contents of a zone.

When an UPDATE is received from non-masters for *slave* zones, we forward the request to the master for that zone, which will presumably update the zone, and then NOTIFY us of the change.

When an UPDATE is received from a master, then we treat it as a short NOTIFY/IXFR transaction. All the same rules apply, such as the requirement that all packets be signed with TSIG keys, and that the change will first be journaled to the disk.

When an UPDATE is received for a *master* zone, then we simply ignore the request, as there is no means available to update zonefiles for *master* zones. However, the request will be fully processed in case a script intervenes.

## Increment Zones Transfer (IXFR)

A zone update is triggered either by a *NOTIFY* from the master server, or by a *refresh* timeout which finds a change in the SOA serial number.

To update a zone, the server will first attempt to establish a TCP connection. If successful, it will then generate an IXFR transaction to update the zone. If IXFR fails, it will backoff and attempt an AXFR. If TCP fails, it will attempt an IXFR request over UDP.

As a secondary master, IXFR transactions are not supported. Therefore, secondary slaves will need to execute full AXFR requests. This is likely to change in the future, as supporting a diff with the last zone change is not expensive.

## Split DNS

Split DNS is no supported, nor is it ever likely to be supported.

## TSIG

Transaction Signatures (TSIG) are required for all dynamic DNS update transactions (UPDATE, NOTIFY, IXFR, AXFR). The key (aka. password) is using the *key {}* statement in the configuration files.

A TSIG key requires two parts, a *name* and a *secret*. Virtually anything can be used as a name and a secret. Most guides for BIND9 describe using the *dnssec-keygen* utility to generate these, but that’s unnecessary.

You could simply make up your own key, such as in the following configuration:

key MyKey {

algorithm HMAC-MD5;

secret "Password"

};

This will work fine, but will be horrible insecure. Transactions can be sniffed from the Internet, and hackers will be able to quickly crack the password. Remember that hackers can execute *billions* of guesses per second, so simply trying all 8 letter combinations can be done in minutes (known as *brute-force* cracking). Worse yet, hackers can do *dictionary* attacks, where they first guess common dictionary words with minor variations, where a “dictionary” consists now only of words in the English dictionary, but other common words people might base passwords on, like “GoBroncos!”.

To defeat dictionary attacks, keys should be chosen completely at random. To defeat brute-force attacks, keys should be long, at least 128-bits. A billion guesses per second of brute force cracking is equivalent to 30 bits being cracked per second, meaning 298 seconds will be needed. A billion years is 230 years. Thus, even with a billion computers, hackers would need longer than the age of the universe to crack 128-bit randomly generated keys.

On any Unix system, the way to do this would be:

$ **dd if=/dev/random bs=1 count=16 | base64**

7zTrFAk8z5YP2IaHNdy0ig==

16+0 records in

16+0 records out

16 bytes (16 B) copied, 6.6499e-05 s, 241 kB/s

The virtual file */dev/random* produces a constant stream of random binary data. The *dd* command can used to extract just the first 16 bytes. The *base64* command will convert this from binary to a string, such as *7zTrFAk8z5YP2IaHNdy0ig==* in this example.

This example uses */dev/random* in this example, but it is only *mostly* random. On Linux, */dev/urandom* is a much more secure choice.

Both the master server on our slave server need to know the same key. Therefore, you are confronted with how to do this securely. If you send over e-mail, it’s likely that evildoers will be able to intercept this. The best way is over SSH, SCP, rsync, or some other similarly secure channel.

Likewise, you should pay attention to whether other users on a system will be able to view the key within configuration files. Since it’s likely only sysadmins have access to your DNS servers anyway, this guide will assume it’s not a significant issue. However, typical BIND9 configuration files are designed so that the *secret* key can be placed in one file that only root can access, while the rest of the configuration files are world-readable, with only the *name* of the key is visible.

Thus, in our example, the configuration we use is the following:

key MyKey {

algorithm HMAC-MD5;

secret "Password"

};

This text could be imported both into master DNS server (which we will assume is a BIND9 server) and the slave server configuration files. If the master server is so configured, it’ll transmit NOTIFY message to the slave, signed by this key, whenever changes happen.

There should probably be a unique key for each slave. Thus, when a hacker breaks into one slave DNS server, they will not be able to modify the zone information for other slaves.

In order to test that the key words, it can be used with the *nsupdate* program, as shown below:

$ **nsupdate –y HMAC-MD5:MyKey:7zTrFAk8z5YP2IaHNdy0ig==**

> **update add www.example.com 600 cname www1.example.com.**

> **send**

The typical hash algorithms are supported (MD5, SHA1, SHA256, SHA512). In addition, custom algorithms can be created with scripts.

## DNSSEC

All DNSSEC records are supported, but none of the DNSSEC functionality. It is assumed that master is responsible for signing zones and creating correct zone data. The slaves simply echo that information.

## Zone information cache

This software supports only one store of zone information: local files in *zonefile* format.

For *slave* zones, zonefiles will be read only one time, upon startup. Updates from the master (UPDATE, IXR, AXFR) will first journaled to the disk (in raw network format), the internal zone information will then be updated, then the original zonefiles updated with the journaled information.

For *master* zones, zonefiles will be read upon startup, on *refresh*, and on *SIGHUP*. The server will never write these zonefiles.

## IPv6 support

IPv6 is fully supported in almost every sense. IPv6 can be used to communicate with the server, on both the *data-plane* and *control-plane* stacks. IPv6 records within zones are handled correctly.

One exception is the deprecated *A6* records from [RFC3363] – only AAAA records are supported.

# Configuration reference

This software uses the same rough configuration format as BIND9. Every effort has been made for backwards compatibility, so some BIND9 configuration files can be used directly. Features this software doesn’t support will either be silently ignored, generate warnings, or cause a hard failure when the configuration file is clearly wanting something important that this software doesn’t support. For example, setting *recursion yes;* will cause a warning message telling the user that this isn’t supported, but will start the server anyway since there is no harm otherwise.

## Basic configuration grammar

Whitespace and comments work as expected.

Newlines are just another form of whitespace, except when ending comments that started with // or #

All comments can start anywhere on a line, even after other text. There are three styles of comments. As is traditional, the character # starts a comment until the end of line. Like C++, the // starts a comment until end-of-line. Like C, the /\* starts a comment that can cross many lines until it ends with \*/. Some examples:

# This is the normal Unix style

// this is the normal C++ style

/\* this is the normal

C style \*/

**token {** # this part is a comment

**token;**

**};**

**token** /\*this is allowed\*/ **token { token; token; };** # so is this

Besides comments and whitespace, the file contains *statements*.

A *statement* is consists of one *token*, optionally followed by additional tokens, optionally followed by additional *statements* within braces, followed by a semicolon. A *token* consists either of something like a DNS name (letters, digits, -, \_, .), or a quoted *string* containing other characters, with double-quotes escaped in the usual manner.

quotes = %x22

token = label / string

label = 1\*(ALPHA / DIGIT / “-“ / “.” / “\_” / “[“ / “]” / “:” / “/”)

string = quotes 0\*(ANY / “\\” / “\””) quotes

statement = 1\*(token) [ **{** 0\*(statement) **}** ] **;**

Some examples:

token;

token "string";

token { token; token { token token; }; };

token "string" token {

token token;

token {

token token;

{ token; token; };

};

token;

};

A common error is to forget the semicolon.

What we refer to as a *top-level* statement is the outer statement containing everything inside. In the above example, there are four top-level statements.

The first step in reading a configuration file is a context-free parse of the top-level statement. The entire statement will be parsed according to this grammar before it’s processed. For performance reasons, each top-level statement is read one at a time, so a configuration file with a million zones doesn’t require being read into a gigabyte’s worth of memory, but instead can be processed one zone at a time.

The second step in processing a top-level is process the contents. The value of the first token determines what sorts of sub-statements can be included. Examples of top-level statements are *options {}*, *zone {}*, *zone-directory {}*, *include*, *key*.

## Global top-level statements

The top-level statements *options* and *zone* are the most important top-level statements, and are described in the own sub-chapters. This sub-chapters describes the remaining ones.

### *include* statement

The include statement looks like the following:

**include “***filename***”;**

This is an include statement of the usual form. Processing of the current file will be halted while the contents of *<filename>* will be loaded.

One reason to use include statements is to separate zone configuration from the rest of configuration information. Zones are frequently updated, whereas the other configuration information is not.

Another reason is for *separation of roles*. The system administrator has more access to the system than lesser administrators. For example, the system administrator may have access to the secret keys, whereas those administering zones do not.

Thus, a configuration file may look like the following:

include "/etc/named/keys.conf"

include "/etc/named/zones.conf"

Thus, the configuration directory may looks something like the following, where *zoneadmins* can write the zone file, read (but not write) the main configuration file, and not read the secret key file:

-rw------- 1 root root 138 Jun 14 00:47 keys.conf

-rw-r--r-- 1 root root 311 Jun 14 00:49 named.conf

-rw-rw-r-- 1 zoneadmins zoneadmins 2382 Jun 14 00:49 zones.conf

### *key* statement

Dynamic updates can only come from trusted sources, with what’s effectively a *username* and *password*. Messages are then signed using this information with *TSIG*. Since human-chosen passwords can easily be cracked, we instead use randomly generated binary keys, and copy/paste the information instead. An example of this configuration is the following:

**key** *username* **{**

**algorithm** *name***;**

**secret** *password***;**

**};**

We describe this as “username” and “password” here, as these are the terms most people are accustomed to. Officially, the username is instead called the “key name”, the “key id”, or even a “key domain name”. Officially, the “password” is known as a “shared secret key”.

The *algorithm* can be one of *HMAC-MD5*, *hmac-sha1*, *hmac-sha256*, or *hmac-sha512*. This parameter is optional. If not set, then HMAC-MD5 will be assumed. Truncated hashes are supported by appending the number of bits with a dash, such as HMAC-MD5-80.

The *secret* is BASE64 encoded. If invalid BASE64 information is included, warning messages will be generated, and the key information will not be loaded, and attempts to use the key will fail. Thus, if you wanted to use “Password1234” as a key, you would first need to BASE64 encode first in order to get UGFzc3dvcmQxMjM0.

This statement must appear at the top level of the configuration file.

For security reasons, you *should* use a separate key for each server you are talking to. That way, if one server in the system gets compromised, the damage will be limited.

## Zone statements

A zone is configured using the top level *zone* statement. Typically, a zone will have its own configuration. If not, it will inherit settings from the top-level *options* statement.

This server also supports a non-standard feature of a *zone-directory*, which contains many (possibly millions) of zonefiles, all sharing the same configuration.

### *type* statement

Two types of zones can be specified, a *master* or a *slave*. The format for this option looks like one of the following two statements:

**type master;**

**type slave;**

For this software, all zones are effectively *slaves*. The ultimate source of all zone information is from a true *master* server, and all changes can be propagated to *secondary slaves*. What this parameter instead configures is the definitive source of zone information.

For a *master* zone, the definitive source is the zonefile on the disk. The server monitors for changes to the zonefile, and will re-read it when it changes, not immediately, but after a time interval. A HUP signal can also force a reload from the zonefile.

For a *slave* zone, the definitive source is the hidden master server via dynamic DNS. The zonefile on disk is optional. If it doesn’t exist, then the server must first fetch the domain via an AXFR from the master. If it does exist on the disk, then the server will load the file, then fetch the SOA record from the master in order to see if it’s up-to-date. The slave will continue to fetch the SOA record on a regular **refresh** in order to monitor for changes, and will also response to dynamic DNS NOTIFY and UPDATE commands from the master. A slave zone must have **masters** configured in this zone, or have a default one set in *options*.

### *file* statement

The *file* statement specifies the location of the zonefile for this zone. This must either be an absolute path starting from root “/”, or a relative path to the current working **directory**.

**file “***filename***”;**

For a *master* zone, this server will never change the file. It is assumed that some other process will edit it, such as a chron job running rsync to transfer updates from the hidden master.

For a *slave* zone, this file is optional (though recommended, a warning will be produced if it’s missing). If it exists, then this server will occasionally modify the file as changes are received over dynamic DNS from the master. The slave will only load this file once at startup, but never read it again.

### *masters* statement

The slave will talk to one or more hidden masters. It can expect to receive UPDATE and NOTIFY packets. It will need to query SOA information on a regular basis, and download the latest changes with IXFR or AXFR. To do this, it needs to know the IP address (or addresses, if more than one), port (which will often be different than 53), and the key needed to talk to the master.

**masters [***name***] [port** *ip\_port***] {**

*ip\_addr* **[port** *ip\_port***] key** *username* **) ;**

**[...]**

**};**

An example of how this might look is the following:

**masters port** *1127* **{**

*10.2.3.15* **key** *master0***;**

*2001:db8:0:1::15* **port** *1128* **key** *master1***;**

**};**

A *zone* or *zone-directory* top-level statement should have a *masters* statement. If they don’t, then they will inherit the settings from the *options* statement.

### *notify*

This controls whether DNS NOTIFY messages are sent to secondary slaves.

**notify yes;**

**notify no;**

**notify master-only;**

**notify explicit;**

If **no** (the default), no message will be sent.

If **explicit**, then only those IP addresses listed in a separate **also-notify** statement will be notified.

If **yes**, then all servers listed in NS records in the zone will be notified, except the master server listed in the SOA record. The SOA master can also be notified with the separate **notify-to-soa** statement.

If **master-only** is specified, only if the zones of **type master** will send such notifications. This is useful when configuring as a default under *options*.

### *also-notify*

Specifies a list of IP address to notify whenever the zone contents change.

**also-notify {**

ipv4 **[port** number**] key [**username**];**

ipv6 **[port** number**] key [**username**];**

**};**

If specified in a *zone* statement, it completely overrides what might have been specified in the *options* statement.

### *notify-to-soa*

By default, notifications (as described in the **notify** statement) are not sent to the SOA master. This changes that behavior. This is in cases where the SOA master specifies a slave when there is a hidden master.

**notify-to-soa yes;**

**notify-to-soa no;**

## *options* Statement

The *options* statement is likely the first one in a configuration file, configuring all the essential options, like which IP addresses and ports the server will use.

**directory “***filename***”**;

**pid-file** “*filename*”;

**listen-on {** *ipv4-address***; }**

**listen-on-v6 {** *ipv6-address***; }**

### *directory*

This specifies the current working directory for the process. Files that don’t start from the root will be relative to this directory. The syntax is:

**directory “***filename***”;**

A typical example of this would look like the following:

**directory “***/var/cache/named***”;**

Unless specified with absolute paths, all zonefiles, statistics, logging, PIDs, and so on will be relative to this directory.

The filename should be an absolute path. Otherwise, the path will be relative to whichever directory the process was started from. Likewise, if this parameter is not set, the default will be “.”, the directory where the process starts.

### *pid-file*

Typical Unix services write their process ID to a file so that other processes can easily find them. An example would be a script that needs to send a HUP signal to the process in order to force a re-load of zonefiles. By default, this service writes to */var/run/named/named.pid*. This can be changed with the *pid-file* option. Its syntax is the following:

**pid-file “***filename***”;**

**pid-file none;**

A typical example of this would look like the following:

**pid-file “***/var/cache/named/robdns.pid***”;**

The contents of this file will consists of a single decimal number, like so

1387

If **none** is specified, then no PID file will be generated.

### *port*

By default, the service listens on port 53 for queries (the *data-plane* port). This port can be changed with this option.

**port** *number***;**

This is useful for testing the service, or for non-standard configurations, such as when using this software as a hidden master (which should never use standard ports).

The **listen-on** option specifying a port should probably be used instead.

### *listen-on, listen-on-v6*

This configures the *data-plane* IP addresses and ports to use for answering NDS queries from the public Internet (see the option **transfer-source** for configuring the *control-plane* address). The syntax is as follows:

**listen-on [port** number**]** **{**

*ip\_address***;**

**!***ip\_address***;**

*ip\_address/cidr***;**

**!***ip\_address/cidr***;**

**any;**

**none;**

**};**

**listen-on-v6 { ... }**

The statement **listen-on** is used for IPv4 addresses, while **listen-on-v6** is used for IPv6. They are otherwise identical.

Multiple statements may be included, and will behave as if combined into one large statement.

**listen-on {**

**192.168.0.167;**

**};**

**listen-on {**

**10.109.36.15;**

**};**

Of special note is when using *wildcard addresses*. For IPv4, a machine may have many IPv4 addresses assigned in a range. Therefore, a CIDR address, such as 192.168.0/24 can be specified. Any IPv4 addresses currently configured on the machine within that range will be used. Some, however, may be excluded. An example might be the following:

**listen-on {**

**192.168.0/24;**

**!192.168.0.25;**

**};**

For IPv6, wildcarding is built-in to the specification, so that a machine may be reachable with billions of IPv6 addresses. It may look like this:

**listen-on-v6 {**

**2001:db8::/96;**

**};**

When using raw sockets, then wildcarding is supported regardless of the machine’s true IP address. In such cases, IPv4 wildcarding works like IPv6, so that if 0/0 is specified as an IPv4 address, the server will respond to queries to any IPv4 address. (This specific configuration is probably a bad idea, because the raw stack will also respond to ARP for all IPv4 addresses).

As expected, **any** will cause the server to use all IP addresses in the system, and **none** will cause it to not use any.

Note that periodically, the IP addresses of the server may change. The server software will potentially open or close new sockets appropriately. This can be set with **interface-interval**.

### *transfer-source, transfer-source-v6, alt-transfer-source, alt-transfer-source-v6*

This specifies the *control-plane* IP addresses to use in order to communicate with hidden masters, such as doing IXFR and AXFR queries. It’s also the IP address we expect hidden masters to use in order to send us UPDATE and NOTIFY packets, rather than the *data-plane* address. The syntax is essentially the same as for the **listen-on** option. If not specified, then the default is to use the *data-plane* IP addresses – unless the data-plane uses raw-sockets, in which case, **any** will become the default.

**transfer-source [port** number**]** **{**

*ip\_address***;**

**!***ip\_address***;**

*ip\_address/cidr***;**

**!***ip\_address/cidr***;**

**any;**

**none;**

**};**

**transfer-source-v6 { ... }**

Remember that the hidden master itself must be configured to use this IP address in its configuration, such as in **also-notify** options.

More than one IP address may be specified. The server will listen on all such addresses for incoming NOTIFY and UPDATE packets. For outgoing connections, it will try the first listed IPv6 addresses first, and if they fail, will backoff and try IPv4 addresses.

### *interface-interval*

The IP addresses of a machine may change over time, as new hardware interfaces as added (such as plugging in USB Ethernet adapters), when DHCP addresses change, or when sysadmins reconfigure IP addresses. Changes are automatically scanned for whenever the configuration file reloads. The system can be configured to scan for such changes automatically as well.

**interface-interface** *minutes***;**

If this option is not specified, the default will be 60, or every hour. The maximum value is 28 days, or 40320 minutes.

If a change is detected, then the server will re-evaluate the configuration set by **listen-on** and **transfer-source** statements in order to close or open sockets as needed.

### *version*

DNS servers can be configured to return their version information upon a lookup for the name “*version.bind*” in the CHAOS class. It’s a little easter egg started by BIND in the 1980s and used by virtually all DNS software since. Should the user want to unique identify this server, then can set the response as follows:

**version “***string***”;**

**version none;**

Note that the license agreement for this server specifies that that “robns” must always be returned in such requests. Thus, any value specified in option will not replace the default “robdns” string, but will instead be appended to it.

You probably do not want to change this option, but instead use the **server-id** or **hostname** options instead.

### *hostname*

DNS servers can be configured to return their hostname with a CHAOS TXT query for *hostname.bind*. By default, this server will return nothing. This option configures something to be sent in the response.

**hostname “***string***”;**

**hostname none;**

**hostname hostname;**

The default is **none**. If the option **hostname** is used, then the name from the *gethostname()* function will be used. Otherwise, if a string is given, that will be given in the response.

### *server-id*

DNS servers can be configured to return a unique string to a CHAOS TXT query for *ID.SERVER*. They will also report then when given an NSID (Name Server Identifier), as specified in [RFC5001]. This is useful when many servers have the same IP address, such as in a load-balanced or anycasted case, so that the individual servers can be disambiguated.

**server-id “***string***”;**

**server-id none;**

**server-id hostname;**

The default is **none**. If the option **hostname** is used, then the name from the *gethostname()* function will be used. Otherwise, if a string is given, that will be given in the response.

### *allow-new-zones*

Traditional DNS servers did not allow automatic zone creation. This server does, and assumes that as the normal operation. Therefore, the default for this parameter (if not specified) is **yes**, and currently the **no** option is not supported and will generate an error.

**allow-new-zones yes;**

Possibly in the future the **no** option will be supported.

### *recursion*

This statement is for producing warning messages. It has two values.

**recursion yes;**

**recursion no;**

If *yes* is specified, then the following message will be generated:

**WARNING: recursion not supported, statement ignored.**

If *no* is specified, then the following message will be generated:

**WARNING: recursion not supported, statement is redundent.**

In either case, this server does not support recursion (aka. resolver, proxying) functionality.

### *auth-nxdomain*

This statement is provided for backwards compatibility, since a lot of examples seem to contain it. The only valid value is **no**, if another is encountered, a warning will be generated.

**auth-nxdomain no;**

## acl Statement

The *acl* statement specifies a named list of IPv4/IPv6 address ranges. Such a list can be used for a lot of purposes, from filtering incoming requests, to listing which IP addresses to notify on zone changes, to configuration which IP addresses the software should use

## zone-directory Statement

One of the intended uses of this software is for large-scale hosting environments which manage a million zones on behalf of customers. In such cases, all the zones will have the same configuration as far as slaves are concerned. Therefore, individual zones do not need separate configuration. This makes dynamic creation and deletion of zones simpler.

The zone-directory statement behaves in most respects like a zone statement, with some specific exceptions. It looks like the following:

**zone-directory “***name***” {**

**filename “***directory***”;** # the directory where files are located

**type** **[***master***|***slave***]**;

**directory-level [***1***|***2***|***3***];** # for subdividing into subdirectories

**directory-refresh [**true**|**false**];**

**};**

If the type is **master**, the default, then on startup, the server will recursively descend that directory and load all zonefiles ending in “.zone”. The zones will be created for each SOA record found. On a refresh or SIGHUP, it will repeat the process, updating zones, creating new zones, or deleting old zones as required.

If the type is **slave**, with dynamic zones, the situation is a little more complicated. As normal, the server will have to first send a query to get the SOA record from the master in order to verify that the zone is up-to-date. Since there may be millions of zones, this may take a long time. Therefore, the **directory-refresh** can be used to inform the master of the timestamp of the latest information. The master can then return a list of all zones which are out-of-date in one response. The server can then update those zones as required.

As NOTIFY packets are received from the server in superslave configuration, the server will create files as necessary.

For a lot of zones, a two-level, or even three-level, directory structure should be used. For static **master** zones, the software will recursively descend any number of levels. For dynamic **slave** zones, it will hash the name and create either one or two levels, each with 100 subdirectories. The hash creates a directory name between 00 and 99. If there are a million zones, this should create at the leaf directories with 100 zones. The **directory-level** statement will specify the number of levels in the hierarchy that will be created.

# Zonefiles

All the DNS information is stored on disk in standard format *zonefiles*. An example is shown below:

$TTL 86400 ; 24 hours

$ORIGIN example.com.

@ 1D IN SOA ns1.example.com. hostmaster.example.com. (

2002022401 ; serial

3H ; refresh

15 ; retry

1w ; expire

3h ; minimum

)

IN NS ns1.example.com. ; in the domain

IN NS ns2.smokeyjoe.com. ; external to domain

IN MX 10 mail.another.com. ; external mail provider

; server host definitions

ns1 IN A 192.168.0.1 ;name server definition

www IN A 192.168.0.2 ;web server definition

ftp IN CNAME www.example.com. ;ftp server definition

; non server domain hosts

bill IN A 192.168.0.3

fred IN A 192.168.0.4

A zone of type *slave* reads the zonefile at startup, but never after that point. It will only write to zonefiles as updates are received from the master. A zone of type *master* will read the zonefile at startup, then re-read the zonefile whenever it changes. It will never write a zonefile.

Only the standard RFC zonefile format is supported. The software doesn’t have a custom binary or compiled format.

## Terminology

An individual piece of information is known as a *resource-record* or *RR*. An RR has a domain-name, a *class*, a *type*, and resource data associated with it. For example, the typical IP address has a class of *IN*, a type of *A*, and four-bytes of resource data containing the IPv4 address.

Multiple records with the same domain-name, class, and type, but different data, are known as an *RRset* or *resource-record set*. Completely duplicate records, including the same data, are not allowed.

All the resource-records in a set will have the same *TTL*. This is not obvious in the zonefile format, which allows different TTLs to be specified for different resource-records, but it’s enforced internally by the code. It’s the entire RRset, rather than individual resource-records, which are the basic unit of DNS.

Multiple RRsets with different types, but the some domain-name, are considered to have the same *owner-name*. Everything at the same *owner-name* is called an *entry* within this software’s code (though that’s not a standard term).

A *zone* contains all the owner-names that have the same suffix as the zone’s SOA record.

The entire database of information, containing all the zones, is known as the *catalog*.

## RR types

Virtually all RR types are supported, even deprecated obsolete ones. It’s just easier supporting all of them than managing a list of those which are, and are not, supported. It is likely, of course, that this software may be one or two RFCs behind the latest, so bleeding edge RR types may not be supported.

For experimentation, scripts can be written in order to parse new RR types.

The TYPE#nnnn is supported for unknown RR types. This will be the format written to zonefiles by slave zones when updates contain unknown resource-records. Likewise, the generic formatting of type information is supported.

## RR limits

The maximum size of a single resource-record is 65535 bytes.

The maximum size of an RRset (resource-record set) is 16-megabytes. There is no maximum number of resource-records in an RRset, except when the size maximum is reached. Note that this maximum includes both information visible to the users as well as internal overhead.

The maximum size of all resource-records at an owner-name is 16-megabytes. The same limits apply as for individual RRsets.

In practice, very large RRsets will cause the server to slow down on updates. That’s because entries are updated by the RCU method, where a complete copy of an owner-name’s RRsets is created during the update.

Putting a hundred thousand A/AAAA records at an owner-name is encouraged. Putting megabytes of data in TXT records at an owner-name is encouraged.

## Zone limits

There is no limit on the size of zones or the numbers of zones.

In practice, the software is designed to load the *.com* zone, which contains over 100 million records. In practice, the software is designed for *DNS hosting*, which may have a million customers, each with their own small zone.

Memory usage is efficient. An 8-gigabyte .com zonefile takes up about 15-gigabytes of memory. A million small zones takes up about 1-gigabytes.

## Journaling

Updates to zones of type *slave* must first be saved in *journal* files to prevent data from being lost/corrupted during a crash. The steps are the following.

First, the updated information is written to the journal, in native wire format (i.e. the same format as an IXFR transaction, just streamed to disk. When the server starts up, after reading a zone, it will check for the journal, and update the zone’s content with the journal before making the information live.

Second, the updates will be applied to the zone in memory, so that queries to the zone will start returning the new information. Note, however, that due to the multi-threaded nature of the software, it may be possible for a response using the old zone information to appear on the wire after a response using the new zone information.

Third, a file with the same name as the zonefile, but ending in “.new” is written to the disk. When the server starts up and cannot find the zonefile, but can find a file of type “.new”, it will remove any journal file, remove any .old file, then remove the “.new” extension (i.e. rename it), then read the zonefile as normal.

Fourth, the existing zonefile is renamed by adding the “.old” extension.

Fifth, the journal file is removed.

Sixth, the “.new” file is renamed to the zonefile, and the “.old” file is removed.

# Live software update/patching

The server is designed for continuous operation, where even applying software patches does not cause downtime. This involves starting a second instance of the server (doubling memory requirements), and then transferring control from the old instance to the new instance.

On operating-systems that support it, this will involve sharing open sockets. When the new server is ready, the sockets will be transferred from one server to the other, without any dropped queries, even under high load.

On operating systems that don’t support it, the old process will have to close open sockets and the new process re-open them as fast as possible. Even in this case, the amount of downtime should be less than transient network congestion.

One concern is that this will double the memory requirements of the server. This is a concern for servering the .com zone which takes up about 15-gigabytes of memory. Servers should expect to have at least 48-gigabytes of memory for such situations.

One concern is updates as slave zones are refreshed with masters. The new process will first notify the old process that it is starting up. The old process will then journal any changes in memory, or even defer updates by a few seconds. The new process will then read the zonefile cache, then notify the old server that it is done. Then, the two instances will synchronize any journaled updates, with the new instance writing those changes to disk as normal.

Once the new instance has caught up, it will then negotiate a hand-off of all the sockets from the old server to the new server, then start responding to queries.

Some updates will be in an incomplete state, such as a slow IXFR/AXFR. The old instance will attempt to continue the operation for several minutes, sending the journaled changes to the new instance (keeping that TCP socket for itself). However, if open transactions stay around for too long without being completed, the old server will simply abort them, notifying the new server to restart the transaction.

Among the set of information the old server transfers to the new server is the *refresh* timeouts of all the slave zones.

A future feature will extend the handoff procedure to work across machines, so that an entire machine can be replaced without downtime. This will involve sending ARPs to notify the router/swtiches of new MAC addresses to redirect traffic.

# Operational security considerations

## Data-plane adapters/addresses

## Control-plane adapters/addresses

## Chroot and setuid

## Dynamic DNS Security

Other systems want to control this IP address.

This server only supports TSIG. TSIG is mandatory in all dynamic DNS packets.

# Features

## Multi-Master

Multi-master mode is supported. Multiple masters can be specified for a zone. If a NOTIFY is received, that IP address will be contacted to receive the update. When a *refresh* expires, the masters will be contacted in-order. If the first doesn’t respond, the second will be tried, and so on.

## SOA Negative Cache TTL [RFC2308]

In zonefiles, the $TTL directive is used to specify the default TTL for records.

The SOA.MINIMUM field is used for negative caching. The SOA record in included in negative responses in order to report that value.

## Limits on number of zones and domains

There are no limits (other than memory).

## Long TXT/SPF strings

In zonefiles, if TXT/SPF strings are longer than 255 characters, they will automatically be split into multiple strings.

## Reduced additional section

DNS often includes “glue” records to speed resolving. However, usually the server is not authoritative for the glue, and thus provide it in the “additional” section of responses rather than the “answer” or “authoritative” sections.

Since the glue cannot be trusted, resolvers are increasingly ignoring the glue, and doing their own separate lookups. Thus, servers should not longer be reporting the glue – unless it can do so authoritatively.

This server does not provide non-authoritative glue.

## NSID [RFC4892, RFC5001]

## Version.bind

Version.server

# Comparisons with other DNS servers

## A brief list of alternatives

### BIND9

This server is patterned after the well-known BIND9 software, which is still the most popular DNS server on the Internet, accounting for more than half of all servers. It is compatible with the same *zonefiles* and *conf* files, and even similar command-line parameters. It is not compatible with *rndc*. Some BIND9 configurations can be used directly with no changes.

This server is about 100 times faster than BIND9. The purpose of this software was to demonstrate the difference in design tradeoffs between a server dedicated to managing a master zone, which BIND9 does very well, compared to one that is exposed directly to the Internet as a slave, which BIND9 does poorly.

### Djbdns TinyDNS

Like TinyDNS, this is a limited-feature server designed specifically to solve security problems.

One major difference is that TinyDNS didn’t solve the scalability problem, and thus fails under DDoS attack.

Another problem is that TinyDNS is user-hostile, supporting non-standard configuration and zonefiles.

Lastly, TinyDNS isn’t “future proof”. It lags current DNS standards by many years. In contrast, this software includes a scripting system that others can use in order to keep the system up-to-date.

### PowerDNS

Like PowerDNS, this server is designed to support a million zones, with *superslave* operation that can create/delete zones on-the-fly.

### KnotDNS

Like KnotDNS, this server focuses on high query rates. However, they are limited by the operating-system’s performance. They go through the normal *control-plane* stack, whereas we use *data-plane* stacks.

### NSD

### ROBDNS/sockets

One benchmark target will be running this software over normal sockets. In this configuration, the operating-system should be the primary bottleneck, and the software should perform close to other packages.

### ROBDNS/libpcap

Another benchmark target will be this software using a custom TCP/IP stack, but over the traditional libpcap/raw-sockets interface. This relies upon the operating system to receive the packets, but avoids the costly network stack.

### ROBDNS/zero-overhead

The intended benchmark for this software uses a custom TCP/IP stack with a zero-overhead driver (DPDK, netmap, PF\_RING). By bypassing the limitations of the operating-system, this should have the highest performance.

## Benchmarks

### Cached query rate

This test sends the same type=A query for *www.example.com* to the server with a small example zone.

The intent of this query is so that servers perform the least amount of work. Most servers should show similar performance on this benchmark, coming close to the operating-system’s inherent packet-rate limits.

### Random TLD query rate

A copy of the *.net* zone with 20 million entries is used, in order to take up several gigabytes of space. Then, random *any* queries are sent using random names. Half the names will exist in the zone, whereas the other half are strings which match no name.

The intent of this benchmark is to measure how well servers handle looking up random information which will not be cached. It should be considerably slower than the non-random query rate.

### Random hosting query rate

A 100,000 zones are created. Random A queries for www.<randomzone> are sent, half of which do in fact exist, and half of which do not.

The performance should be similar to that of the large TLD zone, though it’s possible that a server will do especially bad at this benchmark or the other.

### TLD load time

A copy of the *.net* zone is loaded at startup, with the time measured from startup until the server is ready. This test is run twice.

The first test ignores the time it takes to first compile the zonefile, if that is a separable step. This represents the ideal case for some servers, which can simply memory-map in a compiled file, where startup time roughly equals disk transfer rates. An SSD will be used to optimize this.

The second test includes compilation time. For some servers, this means changing the “backend” to work from zonefiles (like BIND9). For other servers, this means simply including the separable step in their scores (like NSD). Servers will fast zonefile parsers will excel in this test.

The intent is to demonstrate the cost of server failures, where the load time of a large zone can take many minutes, lengthening what would otherwise be a long outage.

### Hosting load time

A 100,000 zones are loaded on startup, with time measured from startup until the server is ready.

Like the previous test, the benchmark is run twice, once to favor software that precompiles the information into a database, and another that favors those who can parse the 100,000 files very quickly.

## Setup

The operating-system is generally tuned as described elsewhere in this document, to optimize the reception of packets on a 10-gbps under the Linux operating system. Some settings may change for different targets in order to maximize the score of a target.

An 8-core, with additional 8-hyperthreads will be used (16-threads total). The core count will be adjusted to maximum the performance of each target, with first the hyperthreads being reduced, then actual cores. It’s likely that many packages will have a limit of about 4 threads, because at this point, the Linux UDP stack starts having its own scalability issues.

# Linux stack tuning

This software is designed to bypass the operating-system and use a custom network stack. However, sometimes it’s necessary to use the operating-system. This section describes how to tune a Linux stack for optimal performance.

There are four areas to optimize: the driver, the TCP/IP stack, general kernel, and the Sockets API. While these are all interrelated, each will be described separately. In addition, there may be *virtual-machine* issues that need to be tweaked.

Linux Ethernet drivers work the same way as other operating systems. The driver allocates a *ring buffer* to receive incoming packets. As packets arrive, the Ethernet hardware uses *DMA* to transfer the packets into the ring buffer, then generates an *interrupt request* (*irq*). The *interrupt handler* does nothing more than schedule a *softirq* (similar to *deferred procedure call* or *DPC* under Windows).

Once the interrupt handler exits, the softirq runs. It’ll have the highest priority on the system, though it can still be preempted by hardware interrupts. It removes the packet from the ring buffer, sends it through the TCP/IP network stack, then puts the packet on the socket’s queue.

Then, when the softirq exits, the application’s process will be run because data has arrived (assuming no other process has higher priority). The application will then call a *sockets* function (like *recvmsg*) in order to pull the packet out of the socket’s queue. The application will then process the packet.

In the meanwhile, other things are happening in the operating system that can interfere with all this, especially on multicore systems.

## Network driver optimization

### IRQ coalescing and NAPI

There is a tradeoff in networking between trying to achieve *low latency* and *high throughput*. Responding immediately to an interrupt is the fastest way to respond to an incoming packet. However, interrupts are costly, so during high network throughput, the system will bog down processing interrupts. To achieve high packet rates, it’s better to process many packets for each interrupt, thereby slowing the response time on average for any particular packet.

The tradeoff, though, is in terms of *microseconds*. Getting sub-microsecond response time is only needed inside the data center. After all, light takes a microsecond to travel 300 meters (a thousand feet). Reasonable latency for DNS is closer to 100 microseconds (30 kilometers distant), or even a full millisecond (300 kilometers distant). Since extreme low latency isn’t a concern, we should instead optimize for high throughput to deal with DDoS attacks.

There are two ways to process multiple packets per interrupt: coalescing and NAPI.

NAPI is usually the default behavior. It processes packets on an interrupt, as normal. However, when incoming interrupt rate becomes too high, it disables interrupts on the card and switches to *polling*. Polling just puts the driver in a tight loop receiving packets as fast as it can. If packet rate is extremely high, it’ll never stop polling, but if packet rates go down, it’ll switch back to interrupts.

Another way, which can be used with NAPI, is throttling the rate of interrupts. Instead of interrupting immediately when a packet arrives, many network adapters can be configured to interrupt no more often than once every 1, 10, or even 100 microseconds.

### RSS

<https://www.kernel.org/doc/Documentation/networking/scaling.txt>

### Net Core

<https://www.kernel.org/doc/Documentation/sysctl/net.txt>

/proc/sys/net/core/netdev\_max\_backlog  
Maximum number of packets to buffer for an interface. Default value is 300. Too much can be bad, leading to buffer-bloat. At high packet rates, this may need to be increased.

/proc/sys/net/core/dev\_weight  
Number of packets to process for each NAPI softirq interrupt. Higher values reduce total CPU usage, but also increase latency for other things being done on the processor. The default value is 64.

### Flow control

Pause packets

## General kernel settings

### irqbalance and manual IRQ assignment

The *irqbalance* daemon runs in the background, reassigning interrupts to different CPUs, depending upon load. The results are highly variable, with benchmark tests varying from 500,000 packets-per-second to 1,500,000 packets-per-second during the same test.

It’s better to manually assign IRQs manually. There are two strategies

### Huge pages

The code relies upon “transparent huge pages”, rather than the older method.

https://www.kernel.org/doc/Documentation/vm/transhuge.txt

Kernel: 2.6.32

echo always >/sys/kernel/mm/transparent\_hugepage/enabled

## Overview

There are two parts to a stack. The first part is the *driver* that receives packets from then network and places them in the kernel. The second part is the *stack* that delivers those packets to a user-mode application reading a socket.

SO\_BUSY\_POLL on a socket

http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/open-source-kernel-enhancements-paper.pdf

<http://www.oracle.com/technetwork/server-storage/vm/ovm3-10gbe-perf-1900032.pdf>

# Compliance

## License

The code is licensed under the *robdns-license.txt*, located in the root directory of the distribution. This is not a GPL or BSD license. The code should not be considered “free” or “open-source”. Any changes submitted via pull requests are given to the author to license as he likes.

The code contains modules written by others. Their code is covered by their licenses. A list of those is provided here.

Lua

MD5 RFC

SHA1 RFC

SHA2 RFC

HMAC RFC

SIP hash

## Export

This code contains cryptographic code, and may require an export license.

## Common Criteria

All security functions are provided by the host environment.

## MISRA C

MISRA is a set of standards for the C programming language when used in critical environments (like factory control systems). To the extent that is reasonable, this code conforms to MISRA guidelines.

### Documented Issues

**Rule 3.2**: only the ASCII 7-bit character set is used.

**Rule 3.3**: no integer division of negative numbers is performed.

**Rule 3.4**: Under Microsoft compilers, the linking #pragma is used in order to link to the WinSock libraries (rather than configure those libraries as a linker option). No other #pragmas are used.

**Rule 3.5**: Bit-field packing is not relied upon.

### Deviations

**Rule 3.6**: This code links to the C runtime libraries (such as libc and pthread) which have not been validated for compliance.

# Glossary

**amplification** – An amplification attack *spoofs* packets from a victim’s IP address to servers on the Internet that reply with bigger responses. This overwhelms a victim, as in a *DDoS* attack. This allows an attack to generate 10 times, or sometimes 100 times, more traffic against a victim. Common protocols used in amplification attacks are DNS, NTP, and SNMP. DNS servers should have *RRL (Response Rate Limiting)* in order to prevent themselves from being exploited as an amplifier.

**authoritative server** – A server that answers queries based upon it’s own database, rather than a *resolver*, *proxy*, or *cache* that answers queries based on sending queries to other sources. An authoritative server may be the *master* (or *primary*) with the original file/database, or a *slave* (or a *secondary*) with a copy from the master.

**authoritative answer** – An answer from an *authoritative server*, rather than an answer from some other server.

**BIND9** – Version 9 of the Berkeley Internet Name Daemon, the oldest and most popular DNS server. More than half of the DNS servers on the Internet are BIND. How BIND handles DNS forms a de facto standard.

**DDoS** – A flood of traffic from thousands of sources (often a botnet) directed at a single victim. The usual goal is simply to overwhelm the network connection, though a secondary goal is to increase processing time on the server. DDoSing DNS servers has for 20 years been a common attack. DDoS attacks may sometimes be *amplified* (see *amplification*).

**delegation** – When a parent *zone* points to a child zone by pointing to the child’s name server (NS record). Delegation is how the hierarchy of zones is glued together. Looking up a name involves starting at the *root zone* and going from delegation to delegation until the final name is found.

**recursive resolver** – A *resolver* that answers an incoming query by making many outgoing queries, as many as are needed to find the final answer. In contrast, a *stub resolver* only sends one query to find the next step, rather than continuing to make queries.

**refresh** – This is the time before the slave server checks the master in order to download any updates to the zone.

**resolver** – A server that, in order to respond to queries, will query other servers itself. A *stub resolver* queries only the next server in the chain while resolving a name. A *recursive resolver* will continue querying other servers until the answer is found.

**Response Rate Limiting** – In order to prevent themselves from being used as an amplifying in an *amplification* attack, DNS servers will limit the responses to a target IP address. The technique is to detect a high rate of large packets sent to a single IP address, and to response with the TQ (truncation) bit, forcing them to use open a TCP connection to the server instead of using UDP.

**RRL** – see *Response Rate Limiting*

**stub resolver** – A type of *resolver* that queries only the next hop in the chain, as opposed to a *recursive resolver* that keeps querying other servers until the answer is found.