dnsdist

Release 1.2.0

PowerDNS.COM BV

Jan 05, 2018

CONTENTS

CHAPTER

ONE

DNSDIST OVERVIEW

dnsdist is a highly DNS-, DoS- and abuse-aware loadbalancer. Its goal in life is to route traffic to the best server, delivering top performance to legitimate users while shunting or blocking abusive traffic.

dnsdist is dynamic, its configuration language is Lua and it can be can be changed at runtime, and its statistics can be queried from a console-like interface or an HTTP API.

A configuration to balance DNS queries to several backend servers:

```
newServer({address="2001:4860:4860::8888", qps=1})
newServer({address="2001:4860:4860::8844", qps=1})
newServer({address="2620:0:ccc::2", qps=10})
newServer({address="2620:0:ccd::2", name="dns1", qps=10})
newServer("192.168.1.2")
setServerPolicy(firstAvailable) -- first server within its QPS limit
```

1.1 Running dnsdist

If you have not worked with dnsdist before, here are some resources to get you going:

- *Install dnsdist*.
- To get a feeling for how it works, see the *Quickstart Guide*.
- *Running and Configuring dnsdist*
- The *Packet Policies* page covers how to apply policies to traffic
- There are several *Guides* about the different features and options
- *Advanced Topics* describes some of the more advanced features
- *Reference Guides* has all the configuration and object information

1.2 Questions, requests or comments?

There are several ways to reach us:

- The dnsdist mailing-list
- #powerdns on irc.oftc.net

If you require commercial support, please see the PowerDNS.com website or email us at powerdns.support.sales@powerdns.com.

This documentation is also available as a PDF document.

CHAPTER

TWO

INSTALLING DNSDIST

dnsdist only runs on UNIX-like systems and there are several ways to install dnsdist. The fastest way is using packages, either from your own operating system vendor or suppied by the PowerDNS project. Building from source is also supported.

2.1 Installing from Packages

If dnsdist is available in your operating system's software repositories, install it from there. However, the version of dnsdist in the repositories might be an older version that might not have a feature that was added in a later version. Or you might want to be brave and try a development snapshot from the master branch. PowerDNS provides software respositories for the most popular distributions. Visit https://repo.powerdns.com for more information and installation instructions.

2.1.1 Debian

For Debian and its derivatives (like Ubuntu) installing the `dnsdist` package should do it:

```
apt-get install -y dnsdist
```

2.1.2 RedHat

For RedHat, CentOS and its derivatives, dnsdist is available in EPEL:

```
yum install -y epel-release
yum install -y dnsdist
```

2.1.3 FreeBSD

dnsdist is also available in FreeBSD ports.

2.2 Installing from Source

In order to compile dnsdist, a modern compiler with C++ 2011 support (like GCC 4.8+ or clang 3.5+) and GNU make are required. dnsdist depends on the following libraries:

- Boost
- Lua 5.1+ or LuaJit
- Editline (libedit)
- libsodium (optional)

- protobuf (optional)
- re2 (optional)

Should **dnsdist** be run on a system with systemd, it is highly recommended to have the systemd header files (`libsystemd-dev` on Debian and `systemd-devel` on CentOS) installed to have **dnsdist** support `systemd-notify`.

2.2.1 From tarball

Release tarballs are available from the downloads site, snapshot and pre-release tarballs can be found as well.

The release tarballs have detached PGP signatures, signed by on these PGP keys:

- FBAE 0323 821C 7706 A5CA 151B DCF5 13FA 7EED 19F3
- 1628 90D0 689D D12D D33E 4696 1C5E E990 D2E7 1575
- B76C D467 1C09 68BA A87D E61C 5E50 715B F2FF E1A7
- 16E1 2866 B773 8C73 976A 5743 6FFC 3343 9B0D 04DF

There is a PGP keyblock with these keys available on http://powerdns.com/powerdns-keyblock.asc.

- Untar the tarball and `cd` into the source directory
- Run `./configure`
- Run `make` or `gmake` (on BSD)

2.2.2 From git

To compile from git, these additional dependencies are required:

- GNU Autoconf
- GNU Automake
- Pandoc
- Ragel

dnsdist sourcecode lives in the PowerDNS git repository but is independent of PowerDNS.

```
git clone https://github.com/PowerDNS/pdns.git
cd pdns/pdns/dnsdistdist
autoreconf -i
./configure
make
```

2.2.3 OS Specific Instructions

None, really.

CHAPTER

THREE

QUICKSTART GUIDE

This guide gives an overview of dnsdist features and operations.

3.1 Running in the Foreground

After *installing* dnsdist, the quickest way to start experimenting is lanching it on the foreground with:

```
dnsdist -l 127.0.0.1:5300 8.8.8.8 2001:4860:4860::8888
```

This will make dnsdist listen on IP address 127.0.0.1, port 5300 and forward all queries to the two listed IP addresses, with a sensible balancing policy.

3.2 `dnsdist` Console and Configuration

Here is more complete configuration, save it to `dnsdist.conf`:

```
newServer({address="2001:4860:4860::8888", qps=1})
newServer({address="2001:4860:4860::8844", qps=1})
newServer({address="2620:0:ccc::2", qps=10})
newServer({address="2620:0:ccd::2", name="dns1", qps=10})
newServer("192.168.1.2")
setServerPolicy(firstAvailable) -- first server within its QPS limit
```

The *newServer()* function is used to add a backend server to the configuration.

Now run dnsdist again, reading this configuration:

```
$ dnsdist -C dnsdist.conf --local=0.0.0.0:5300
Marking downstream [2001:4860:4860::8888]:53 as 'up'
Marking downstream [2001:4860:4860::8844]:53 as 'up'
Marking downstream [2620:0:ccc::2]:53 as 'up'
Marking downstream [2620:0:ccd::2]:53 as 'up'
Marking downstream 192.168.1.2:53 as 'up'
Listening on 0.0.0.0:5300
>
```

You can now send queries to port 5300, and get answers:

```
$ dig -t aaaa powerdns.com @127.0.0.1 -p 5300 +short
2001:888:2000:1d::2
```

Note that dnsdist dropped us in a prompt above, where we can get some statistics:

```
> showServers()
#   Address                 State     Qps    Qlim Ord Wt    Queries   Drops␣
↪Drate   Lat Pools
```

```
0   [2001:4860:4860::8888]:53     up      0.0       1   1  1           1       0   0.
↪0    0.0
1   [2001:4860:4860::8844]:53     up      0.0       1   1  1           0       0   0.
↪0    0.0
2   [2620:0:ccc::2]:53            up      0.0      10   1  1           0       0   0.
↪0    0.0
3   [2620:0:ccd::2]:53            up      0.0      10   1  1           0       0   0.
↪0    0.0
4   192.168.1.2:53                up      0.0       0   1  1           0       0   0.
↪0    0.0
All                                       0.0                           1       0
```

showServers() is usually one of the first commands you will use when logging into the console. More advanced topics are covered in *Working with the dnsdist Console*.

Here we also see our configuration. 5 downstream servers have been configured, of which the first 4 have a QPS limit (of 1, 1, 10 and 10 queries per second, respectively).

The final server has no limit, which we can easily test:

```
$ for a in {0..1000}; do dig powerdns.com @127.0.0.1 -p 5300 +noall > /dev/null;␣
↪done
```

```
> showServers()
#   Address                    State      Qps    Qlim Ord Wt    Queries   Drops␣
↪Drate   Lat Pools
0   [2001:4860:4860::8888]:53     up      1.0       1   1  1           7       0   0.
↪0    1.6
1   [2001:4860:4860::8844]:53     up      1.0       1   1  1           6       0   0.
↪0    0.6
2   [2620:0:ccc::2]:53            up     10.3      10   1  1          64       0   0.
↪0    2.4
3   [2620:0:ccd::2]:53            up     10.3      10   1  1          63       0   0.
↪0    2.4
4   192.168.1.2:53                up    125.8       0   1  1         671       0   0.
↪0    0.4
All                                     145.0                         811       0
```

Note that the first 4 servers were all limited to near their configured QPS, and that our final server was taking up most of the traffic. No queries were dropped, and all servers remain up.

3.2.1 Changing Server Settings

The servers from *showServers()* are numbered, *getServer()* is used to get this *Server* object to manipulate it.

To force a server down, try *Server:setDown()*:

```
> getServer(0):setDown()
> showServers()
#   Address                    State      Qps    Qlim Ord Wt    Queries   Drops␣
↪Drate   Lat Pools
0   [2001:4860:4860::8888]:53   DOWN      0.0       1   1  1           8       0   0.
↪0    0.0
...
```

The DOWN in all caps means it was forced down. A lower case down would've meant that dnsdist itself had concluded the server was down. Similarly, *Server:setUp()* forces a server to be up, and *Server:setAuto()* returns it to the default availability-probing.

To change the QPS for a server, use *Server:setQPS()*:

```
> getServer(0):setQPS(1000)
```

3.3 Restricting Access

By default, dnsdist listens on `127.0.0.1` (not `::1`!), port 53.

To listen on a different address, use the `-l` command line option (useful for testing in the foreground), or use *`setLocal()`* and *`addLocal()`* in the configuration file:

```
setLocal('192.0.2.53')       -- Listen on 192.0.2.53, port 53
addLocal('192.0.2.54:5300') -- Also listen on 192.0.2.54, port 5300
```

Before packets are processed they have to pass the ACL, which helpfully defaults to **RFC 1918** private IP space. This prevents us from easily becoming an open DNS resolver.

Adding network ranges to the *ACL* is done with the *`setACL()`* and *`addACL()`* functions:

```
setACL({'192.0.2.0/28', '2001:DB8:1::/56'}) -- Set the ACL to only allow these␣
↪subnets
addACL('2001:DB8:2::/56')                   -- Add this subnet to the existing ACL
```

3.4 More Information

Following this quickstart guide allowed you to set up a basic balancing dnsdist instance. However, dnsdist is much more powerful. See the *Guides* and/or the *Advanced Topics* sections on how to shape, shut and otherwise manipulate DNS traffic.

CHAPTER

FOUR

RUNNING AND CONFIGURING DNSDIST

dnsdist is meant to run as a daemon. As such, distribution native packages know how to stop/start themselves using operating system services.

It is configured with a configuration file called `dnsdist.conf` The default path to this file is determined by the `SYSCONFDIR` variable during compilation. Most likely this path is `/etc/dnsdist`, `/etc` or `/usr/local/etc/`, dnsdist will tell you on startup which file it reads.

dnsdist is designed to (re)start almost instantly. But to prevent downtime when changing configuration, the console (see *Working with the dnsdist Console*) can be used for live configuration.

Issueing `delta()` on the console will print the changes to the configuration that have been made since startup:

```
> delta()
-- Wed Feb 22 2017 11:31:44 CET
addLocal('127.0.0.1:5301', false)
-- Wed Feb 22 2017 12:03:48 CET
addACL('2.0.0.0/8')
-- Wed Feb 22 2017 12:03:50 CET
addACL('2.0.0.0/8')
-- Wed Feb 22 2017 12:03:51 CET
addACL('2.0.0.0/8')
-- Wed Feb 22 2017 12:05:51 CET
addACL('2001:db8::1')
```

These commands can be copied to the configuration file, should they need to persist after a restart.

4.1 Running as unprivileged user

dnsdist can drop privileges using the `--uid` and `--gid` command line switches to ensure it does not run with root privileges. Note that **dnsdist** drops its privileges **after** parsing its startup configuration and binding its listening and initial *newServer()* sockets as user *root*. It is highly recommended to create a system user and group for **dnsdist**. Note that most packaged versions of **dnsdist** already create this user.

CHAPTER

FIVE

PACKET POLICIES

dnsdist works in essence like any other loadbalancer:

It receives packets on one or several addresses it listens on, and determines whether it will process this packet based on the *Access Control*. Should the packet be processed, dnsdist attempts to match any of the configured rules in order and when one matches, the associated action is performed.

These rule and action combinations are considered policies.

5.1 Packet Actions

Each packet can be:

- Dropped
- Turned into an answer directly
- Forwarded to a downstream server
- Modified and forwarded to a downstream and be modified back
- Be delayed

This decision can be taken at different times during the forwarding process.

5.1.1 Examples

Rules for traffic exceeding QPS limits

Traffic that exceeds a QPS limit, in total or per IP (subnet) can be matched by a rule.

For example:

```
addAction(MaxQPSIPRule(5, 32, 48), DelayAction(100))
```

This measures traffic per IPv4 address and per /48 of IPv6, and if traffic for such an address (range) exceeds 5 qps, it gets delayed by 100ms.

As another example:

```
addAction(MaxQPSIPRule(5), NoRecurseAction())
```

This strips the Recursion Desired (RD) bit from any traffic per IPv4 or IPv6 /64 that exceeds 5 qps. This means any those traffic bins is allowed to make a recursor do 'work' for only 5 qps.

If this is not enough, try:

```
addAction(MaxQPSIPRule(5), DropAction())
```

or:

```
addAction(MaxQPSIPRule(5), TCAction())
```

This will respectively drop traffic exceeding that 5 QPS limit per IP or range, or return it with TC=1, forcing clients to fall back to TCP.

To turn this per IP or range limit into a global limit, use `NotRule(MaxQPSRule(5000))` instead of *`MaxQPSIPRule()`*.

Regular Expressions

`RegexRule()` matches a regular expression on the query name, and it works like this:

```
addAction(RegexRule("[0-9]{5,}"), DelayAction(750)) -- milliseconds
addAction(RegexRule("[0-9]{4,}\\.example$"), DropAction())
```

This delays any query for a domain name with 5 or more consecutive digits in it. The second rule drops anything with more than 4 consecutive digits within a .example domain.

Note that the query name is presented without a trailing dot to the regex. The regex is applied case insensitively.

Alternatively, if compiled in, *`RE2Rule()`* provides similar functionality, but against libre2.

5.2 Rule Generators

`dnsdist` contains several functions that make it easier to add actions and rules.

`addAnyTCRule`()
Deprecated since version 1.2.0.

Set the TC-bit (truncate) on ANY queries received over UDP, forcing a retry over TCP. This function is deprecated as of 1.2.0 and will be removed in 1.3.0. This is equivalent to doing:

```
addAction(AndRule({QTypeRule(dnsdist.ANY), TCPRule(false)}), TCAction())
```

`addDelay`(*DNSrule*, *delay*)
Deprecated since version 1.2.0.

Delay the query for `delay` milliseconds before sending to a backend. This function is deprecated as of 1.2.0 and will be removed in 1.3.0, please use instead:

addAction(DNSRule, DelayAction(delay))

Parameters

- **`DNSRule`** – The DNSRule to match traffic
- **`delay`** (*int*) – The delay time in milliseconds.

`addDisableValidationRule`(*DNSrule*)
Deprecated since version 1.2.0.

Set the CD (Checking Disabled) flag to 1 for all queries matching the DNSRule. This function is deprecated as of 1.2.0 and will be removed in 1.3.0. Please use the *`DisableValidationAction()`* action instead.

`addDomainBlock`(*domain*)
Deprecated since version 1.2.0.

Drop all queries for `domain` and all names below it. Deprecated as of 1.2.0 and will be removed in 1.3.0, please use instead:

addAction(domain, DropAction())

Parameters **domain** (*string*) – The domain name to block

addDomainSpoof (*domain*, *IPv4*[, *IPv6*])
addDomainSpoof (*domain, {IP[,...]}*)
Deprecated since version 1.2.0.

Generate answers for A/AAAA/ANY queries. This function is deprecated as of 1.2.0 and will be removed in 1.3.0, please use:

addAction(domain, SpoofAction({IP[,...]}))

or:

addAction(domain, SpoofAction(IPv4[, IPv6]))

Parameters

- **domain** (*string*) – Domain name to spoof for
- **IPv4** (*string*) – IPv4 address to spoof in the reply
- **IPv6** (*string*) – IPv6 address to spoof in the reply
- **IP** (*string*) – IP address to spoof in the reply

addDomainCNAMESpoof (*domain*, *cname*)
Deprecated since version 1.2.0.

Generate CNAME answers for queries. This function is deprecated as of 1.2.0 and will be removed in 1.3.0, in favor of using:

addAction(domain, SpoofCNAMEAction(cname))

Parameters

- **domain** (*string*) – Domain name to spoof for
- **cname** (*string*) – Domain name to add CNAME to

addLuaAction (*DNSrule*, *function*)
Invoke a Lua function that accepts a *DNSQuestion*. This function works similar to using *LuaAction()*.

The `function` should return a *DNSAction*.

Parameters

- **DNSRule** – match queries based on this rule
- **function** (*string*) – the name of a Lua function

addLuaResponseAction (*DNSrule*, *function*)
Invoke a Lua function that accepts a *DNSQuestion* on the response. This function works similar to using *LuaAction()*.

The `function` should return a *DNSAction*.

Parameters

- **DNSRule** – match queries based on this rule
- **function** (*string*) – the name of a Lua function

addNoRecurseRule (*DNSrule*)
Deprecated since version 1.2.0.

Clear the RD flag for all queries matching the rule. This function is deprecated as of 1.2.0 and will be removed in 1.3.0, please use:

addAction(DNSRule, NoRecurseAction())

Parameters **DNSRule** – match queries based on this rule

addPoolRule (*DNSRule*, *pool*)
Deprecated since version 1.2.0.

Send queries matching the first argument to the pool `pool`. e.g.:

```
addPoolRule("example.com", "myPool")
```

This function is deprecated as of 1.2.0 and will be removed in 1.3.0, this is equivalent to:

```
addAction("example.com", PoolAction("myPool"))
```

Parameters

- **DNSRule** – match queries based on this rule
- **pool** (*string*) – The name of the pool to send the queries to

addQPSLimit (*DNSrule*, *limit*)
Deprecated since version 1.2.0.

Limit queries matching the DNSRule to `limit` queries per second. All queries over the limit are dropped. This function is deprecated as of 1.2.0 and will be removed in 1.3.0, please use:

addAction(DNSRule, QPSAction(limit))

Parameters

- **DNSRule** – match queries based on this rule
- **limit** (*int*) – QPS limit for this rule

addQPSPoolRule (*DNSRule*, *limit*, *pool*)
Deprecated since version 1.2.0.

Send at most `limit` queries/s for this pool, letting the subsequent rules apply otherwise. This function is deprecated as of 1.2.0 and will be removed in 1.3.0, as it is only a convience function for the following syntax:

```
addAction("192.0.2.0/24", QPSPoolAction(15, "myPool")
```

Parameters

- **DNSRule** – match queries based on this rule
- **limit** (*int*) – QPS limit for this rule
- **pool** (*string*) – The name of the pool to send the queries to

5.3 Managing Rules

Active Rules can be shown with *showRules()* and removed with *rmRule()*:

```
> addAction("h4xorbooter.xyz.", QPSAction(10))
> addAction({"130.161.0.0/16", "145.14.0.0/16"} , QPSAction(20))
> addAction({"nl.", "be."}, QPSAction(1))
> showRules()
#     Matches Rule                                                Action
0           0 h4xorbooter.xyz.                                    qps limit to 10
1           0 130.161.0.0/16, 145.14.0.0/16                       qps limit to 20
2           0 nl., be.                                            qps limit to 1
```

For Rules related to the incoming query:

addAction (*DNSrule*, *action*)
Add a Rule and Action to the existing rules.

Parameters

- **rule** (*DNSrule*) – A DNSRule, e.g. an `allRule()` or a compounded bunch of rules using e.g. `AndRule()`
- **action** – The action to take

clearRules ()
Remove all current rules.

getAction (*n*) → Action
Returns the Action associated with rule `n`.

Parameters **n** (*int*) – The rule number

mvRule (*from*, *to*)
Move rule `from` to a position where it is in front of `to`. `to` can be one larger than the largest rule, in which case the rule will be moved to the last position.

Parameters

- **from** (*int*) – Rule number to move
- **to** (*int*) – Location to more the Rule to

newRuleAction (*rule*, *action*)
Return a pair of DNS Rule and DNS Action, to be used with *setRules()*.

Parameters

- **rule** (*Rule*) – A *Rule*
- **action** (*Action*) – The *Action* to apply to the matched traffic

setRules (*rules*)
Replace the current rules with the supplied list of pairs of DNS Rules and DNS Actions (see *newRuleAction()*)

Parameters **rules** (*[RuleAction]*) – A list of RuleActions

showRules ()
Show all defined rules for queries.

topRule ()
Move the last rule to the first position.

rmRule (*n*)
Remove rule `n`.

Parameters **n** (*int*) – Rule number to remove

For Rules related to responses:

addResponseAction (*DNSRule*, *action*)
Add a Rule and Action for responses to the existing rules.

Parameters

- **DNSRule** – A DNSRule, e.g. an `allRule()` or a compounded bunch of rules using e.g. `AndRule()`
- **action** – The action to take

mvResponseRule (*from*, *to*)
Move response rule `from` to a position where it is in front of `to`. `to` can be one larger than the largest rule, in which case the rule will be moved to the last position.

Parameters

- **from** (*int*) – Rule number to move
- **to** (*int*) – Location to more the Rule to

rmResponseRule(*n*)
Remove response rule `n`.

Parameters **n** (*int*) – Rule number to remove

showResponseRules()
Show all defined response rules.

topResponseRule()
Move the last response rule to the first position.

Functions for manipulation Cache Hit Rules:

addCacheHitAction(*DNSRule*, *action*)
New in version 1.2.0.

Add a Rule and Action for Cache Hits to the existing rules.

Parameters

- **DNSRule** – A DNSRule, e.g. an `allRule()` or a compounded bunch of rules using e.g. `AndRule()`
- **action** – The action to take

mvCacheHitResponseRule(*from*, *to*)
New in version 1.2.0.

Move cache hit response rule `from` to a position where it is in front of `to`. `to` can be one larger than the largest rule, in which case the rule will be moved to the last position.

Parameters

- **from** (*int*) – Rule number to move
- **to** (*int*) – Location to more the Rule to

rmCacheHitResponseRule(*n*)
New in version 1.2.0.

Remove cache hit response rule `n`.

Parameters **n** (*int*) – Rule number to remove

showCacheHitResponseRules()
New in version 1.2.0.

Show all defined cache hit response rules.

topCacheHitResponseRule()
New in version 1.2.0.

Move the last cache hit response rule to the first position.

5.4 Matching Packets (Selectors)

Packets can be matched by selectors, called a `DNSRule`. These `DNSRules` be one of the following items:

- A string that is either a domain name or netmask
- A list of strings that are either domain names or netmasks
- A *DNSName*

- A list of *DNSName*s
- A (compounded) `Rule`

New in version 1.2.0: A DNSRule can also be a *DNSName* or a list of these

AllRule()
Matches all traffic

DNSSECRule()
Matches queries with the DO flag set

MaxQPSIPRule(*qps*[, *v4Mask*[, *v6Mask*[, *burst*]]])
Matches traffic for a subnet specified by `v4Mask` or `v6Mask` exceeding `qps` queries per second up to `burst` allowed

Parameters

- **qps** (*int*) – The number of queries per second allowed, above this number traffic is matched
- **v4Mask** (*int*) – The IPv4 netmask to match on. Default is 32 (the whole address)
- **v6Mask** (*int*) – The IPv6 netmask to match on. Default is 64
- **burst** (*int*) – The number of burstable queries per second allowed. Default is same as qps

MaxQPSRule(*qps*)
Matches traffic exceeding this qps limit. If e.g. this is set to 50, starting at the 51st query of the current second traffic is matched. This can be used to enforce a global QPS limit.

Parameters **qps** (*int*) – The number of queries per second allowed, above this number traffic is matched

NetmaskGroupRule(*nmg*[, *src*])
Matches traffic from/to the network range specified in `nmg`.

Set the `src` parameter to false to match `nmg` against destination address instead of source address. This can be used to differentiate between clients

Parameters

- **nmg** (*NetMaskGroup*) – The NetMaskGroup to match on
- **src** (*bool*) – Whether to match source or destination address of the packet. Defaults to true (matches source)

OpcodeRule(*code*)
Matches queries with opcode `code`. `code` can be directly specified as an integer, or one of the *built-in DNSOpcodes <DNSOpcode>*.

Parameters **code** (*int*) – The opcode to match

ProbaRule(*probability*)
New in version 1.3.0.

Matches queries with a given probability. 1.0 means "always"

Parameters **probability** (*double*) – Probability of a match

QClassRule(*qclass*)
Matches queries with the specified `qclass`. `class` can be specified as an integer or as one of the built-in *QClass*.

Parameters **qclass** (*int*) – The Query Class to match on

QNameRule(*qname*)
New in version 1.2.0: Matches queries with the specified qname exactly.

param string qname Qname to match

QNameLabelsCountRule(*min*, *max*)
Matches if the qname has less than `min` or more than `max` labels.

Parameters

- **min** (*int*) – Minimum number of labels
- **max** (*int*) – Maximum nimber of labels

QNameWireLengthRule(*min*, *max*)
Matches if the qname's length on the wire is less than `min` or more than `max` bytes.

Parameters

- **min** (*int*) – Minimum number of bytes
- **max** (*int*) – Maximum nimber of bytes

QTypeRule(*qtype*)
Matches queries with the specified `qtype` `qtype` may be specified as an integer or as one of the built-in QTypes. For instance `dnsdist.A`, `dnsdist.TXT` and `dnsdist.ANY`.

Parameters **qtype** (*int*) – The QType to match on

RCodeRule(*rcode*)
Matches queries or responses the specified `rcode`. `rcode` can be specified as an integer or as one of the built-in *RCode <DNSRcode>*.

Parameters **rcode** (*int*) – The RCODE to match on

RDRule()
New in version 1.2.0.

Matches queries with the RD flag set.

RegexRule(*regex*)
Matches the query name against the `regex`.

```
addAction(RegexRule("[0-9]{5,}"), DelayAction(750)) -- milliseconds
addAction(RegexRule("[0-9]{4,}\\.example$"), DropAction())
```

This delays any query for a domain name with 5 or more consecutive digits in it. The second rule drops anything with more than 4 consecutive digits within a .EXAMPLE domain.

Note that the query name is presented without a trailing dot to the regex. The regex is applied case insensitively.

Parameters **regex** (*string*) – A regular expression to match the traffic on

RecordsCountRule(*section*, *minCount*, *maxCount*)
Matches if there is at least `minCount` and at most `maxCount` records in the section `section`. `section` can be specified as an integer or as a *DNS Section*.

Parameters

- **section** (*int*) – The section to match on
- **minCount** (*int*) – The minimum number of entries
- **maxCount** (*int*) – The maximum number of entries

RecordsTypeCountRule(*section*, *qtype*, *minCount*, *maxCount*)
Matches if there is at least `minCount` and at most `maxCount` records of type `type` in the section `section`. `section` can be specified as an integer or as a ref:*DNSSection*. `qtype` may be specified as an integer or as one of the built-in QTypes, for instance `dnsdist.A` or `dnsdist.TXT`.

Parameters

- **section** (*int*) – The section to match on
- **qtype** (*int*) – The QTYPE to match on

- **minCount** (*int*) – The minimum number of entries
- **maxCount** (*int*) – The maximum number of entries

RE2Rule (*regex*)
Matches the query name against the supplied regex using the RE2 engine.

For an example of usage, see *RegexRule()*.

Note Only available when dnsdist was built with libre2 support.

Parameters **regex** (*str*) – The regular expression to match the QNAME.

SuffixMatchNodeRule (*smn*[, *quiet*])
Matches based on a group of domain suffixes for rapid testing of membership. Pass true as second parameter to prevent listing of all domains matched.

Parameters

- **smb** (*SuffixMatchNode*) – The SuffixMatchNode to match on
- **quiet** (*bool*) – Do not return the list of matched domains. Default is false.

TagRule (*name*[, *value*])
Matches question or answer with a tag named `name` set. If `value` is specified, the existing tag value should match too.

Parameters

- **name** (*bool*) – The name of the tag that has to be set
- **value** (*bool*) – If set, the value the tag has to be set to. Default is unset

TCPRule ([*tcp*])
Matches question received over TCP if `tcp` is true, over UDP otherwise.

Parameters **tcp** (*bool*) – Match TCP traffic. Default is true.

TrailingDataRule ()
Matches if the query has trailing data.

5.4.1 Combining Rules

andRule (*selectors*)
Matches traffic if all `selectors` match.

Parameters **selectors** (*{Rule}*) – A table of Rules

NotRule (*selector*)
Matches the traffic if the `selector` rule does not match;

Parameters **selector** (*Rule*) – A Rule

OrRule (*selectors*)
Matches the traffic if one or more of the the `selectors` Rules does match.

Parameters **selector** (*{Rule}*) – A table of Rules

5.4.2 Convience Functions

makeRule (*rule*)
Make a *NetmaskGroupRule()* or a *SuffixMatchNodeRule()*, depending on it is called. `makeRule("0.0.0.0/0")` will for example match all IPv4 traffic, `makeRule({"be","nl","lu"})` will match all Benelux DNS traffic.

Parameters **rule** (*string*) – A string to convert to a rule.

5.5 Actions

Matching Packets (Selectors) need to be combined with an action for them to actually do something with the matched packets. Some actions allow further processing of rules, this is noted in their description. The following actions exist.

AllowAction()
Let these packets go through.

AllowResponseAction()
Let these packets go through.

DelayAction(*milliseconds*)
Delay the response by the specified amount of milliseconds (UDP-only). Subsequent rules are processed after this rule.

Parameters **milliseconds** (*int*) – The amount of milliseconds to delay the response

DelayResponseAction(*milliseconds*)
Delay the response by the specified amount of milliseconds (UDP-only). Subsequent rules are processed after this rule.

Parameters **milliseconds** (*int*) – The amount of milliseconds to delay the response

DisableECSAction()
Disable the sending of ECS to the backend. Subsequent rules are processed after this rule.

DisableValidationAction()
Set the CD bit in the query and let it go through.

DropAction()
Drop the packet.

DropResponseAction()
Drop the packet.

ECSOverrideAction(*override*)
Whether an existing EDNS Client Subnet value should be overridden (true) or not (false). Subsequent rules are processed after this rule.

Parameters **override** (*bool*) – Whether or not to override ECS value

ECSPrefixLengthAction(*v4*, *v6*)
Set the ECS prefix length. Subsequent rules are processed after this rule.

Parameters

- **v4** (*int*) – The IPv4 netmask length
- **v6** (*int*) – The IPv6 netmask length

LogAction([*filename*[, *binary*[, *append*[, *buffered*]]]])
Log a line for each query, to the specified `file` if any, to the console (require verbose) otherwise. When logging to a file, the `binary` optional parameter specifies whether we log in binary form (default) or in textual form. The `append` optional parameter specifies whether we open the file for appending or truncate each time (default). The `buffered` optional parameter specifies whether writes to the file are buffered (default) or not. Subsequent rules are processed after this rule.

Parameters

- **filename** (*string*) – File to log to
- **binary** (*bool*) – Do binary logging. Default true
- **append** (*bool*) – Append to the log. Default false
- **buffered** (*bool*) – Use buffered I/O. default true

LuaAction (*function*)
Invoke a Lua function that accepts a *DNSQuestion*.

The `function` should return a *DNSAction*.

Parameters **function** (*string*) – the name of a Lua function

LuaResponseAction (*function*)
Invoke a Lua function that accepts a *DNSResponse*.

The `function` should return a DNSResponseAction.

Parameters **function** (*string*) – the name of a Lua function

MacAddrAction (*option*)
Add the source MAC address to the query as EDNS0 option `option`. This action is currently only supported on Linux. Subsequent rules are processed after this rule.

Parameters **option** (*int*) – The EDNS0 option number

NoneAction ()
Does nothing. Subsequent rules are processed after this rule.

NoRecurseAction ()
Strip RD bit from the question, let it go through. Subsequent rules are processed after this rule.

PoolAction (*poolname*)
Send the packet into the specified pool.

Parameters **poolname** (*string*) – The name of the pool

QPSAction (*maxqps*)
Drop a packet if it does exceed the `maxqps` queries per second limits. Letting the subsequent rules apply otherwise.

Parameters **maxqps** (*int*) – The QPS limit

QPSPoolAction (*maxqps*, *poolname*)
Send the packet into the specified pool only if it does not exceed the `maxqps` queries per second limits. Letting the subsequent rules apply otherwise.

Parameters

- **maxqps** (*int*) – The QPS limit for that pool
- **poolname** (*string*) – The name of the pool

RCodeAction (*rcode*)
Reply immediatly by turning the query into a response with the specified `rcode`. `rcode` can be specified as an integer or as one of the built-in *RCode*.

Parameters **rcode** (*int*) – The RCODE to respond with.

RemoteLogAction (*remoteLogger*[, *alterFunction*])
Send the content of this query to a remote logger via Protocol Buffer. `alterFunction` is a callback, receiving a *DNSQuestion* and a *DNSDistProtoBufMessage*, that can be used to modify the Protocol Buffer content, for example for anonymization purposes

Parameters

- **remoteLogger** (*string*) – The *remoteLogger* object to write to
- **alterFunction** (*string*) – Name of a function to modify the contents of the logs before sending

RemoteLogResponseAction (*remoteLogger*[, *alterFunction*[, *includeCNAME*]])
Send the content of this response to a remote logger via Protocol Buffer. `alterFunction` is the same callback that receiving a *DNSQuestion* and a *DNSDistProtoBufMessage*, that can be used to modify the Protocol Buffer content, for example for anonymization purposes `includeCNAME` indicates whether

CNAME records inside the response should be parsed and exported. The default is to only exports A and AAAA records

Parameters

- **remoteLogger** (*string*) – The *remoteLogger* object to write to
- **alterFunction** (*string*) – Name of a function to modify the contents of the logs before sending
- **includeCNAME** (*bool*) – Whether or not to parse and export CNAMEs. Default false

SkipCacheAction()
Don't lookup the cache for this query, don't store the answer.

SNMPTrapAction([*message*])
Send an SNMP trap, adding the optional `message` string as the query description. Subsequent rules are processed after this rule.

Parameters **message** (*string*) – The message to include

SNMPTrapResponseAction([*message*])
Send an SNMP trap, adding the optional `message` string as the query description. Subsequent rules are processed after this rule.

Parameters **message** (*string*) – The message to include

SpoofAction(*ip[, ip[...]]*)
SpoofAction(*ips*)
Forge a response with the specified IPv4 (for an A query) or IPv6 (for an AAAA) addresses. If you specify multiple addresses, all that match the query type (A, AAAA or ANY) will get spoofed in.

Parameters

- **ip** (*string*) – An IPv4 and/or IPv6 address to spoof
- **ips** (*{string}*) – A table of IPv4 and/or IPv6 addresses to spoof

SpoofCNAMEAction(*cname*)
Forge a response with the specified CNAME value.

Parameters **cname** (*string*) – The name to respond with

TagAction(*name*, *value*)
Associate a tag named `name` with a value of `value` to this query, that will be passed on to the response.

Parameters

- **name** (*string*) – The name of the tag to set
- **cname** (*string*) – The value of the tag

TagResponseAction(*name*, *value*)
Associate a tag named `name` with a value of `value` to this response.

Parameters

- **name** (*string*) – The name of the tag to set
- **cname** (*string*) – The value of the tag

TCAction()
Create answer to query with TC and RD bits set, to force the client to TCP.

TeeAction(*remote*[, *addECS*])
Send copy of query to `remote`, keep stats on responses. If `addECS` is set to true, EDNS Client Subnet information will be added to the query.

Parameters

- **remote** (*string*) – An IP:PORT conbination to send the copied queries to

- **addECS** (*bool*) – Whether or not to add ECS information. Default false

CHAPTER

SIX

STATISTICS

dnsdist keeps statistics on the queries is receives and send out. They can be accessed in different ways:

- via the console (see *Working with the dnsdist Console*), using `dumpStats()` for the general ones, `showServers()` for the ones related to the backends, `showBinds()` for the frontends, *getPool("pool name"):getCache():printStats()* for the ones related to a specific cache and so on
- via the internal webserver (see *Built-in webserver*)
- via Carbon / Graphite / Metronome export (see *Exporting statistics via Carbon*)
- via SNMP (see *SNMP support*)

6.1 acl-drops

The number of packets dropped bacause of the *ACL*.

6.2 cache-hits

Number of times an answer was retrieved from *cache*.

6.3 cache-misses

Number of times an answer was not found in the *cache*.

6.4 cpu-sys-msec

Milliseconds spent by **dnsdist** in the "system" state.

6.5 cpu-user-msec

Milliseconds spent by **dnsdist** in the "user" state.

6.6 downstream-send-errors

Number of errors when sending a query to a backend.

6.7 downstream-timeouts

Number of queries not answer in time by a backend.

6.8 dyn-block-nmg-size

Number of dynamic blocks entries.

6.9 dyn-blocked

Number of queries dropped because of a dynamic block.

6.10 empty-queries

Number of empty queries received from clients.

6.11 fd-usage

Number of currently used file descriptors.

6.12 latency-avg100

Average response latency in microseconds of the last 100 packets

6.13 latency-avg1000

Average response latency in microseconds of the last 1000 packets.

6.14 latency-avg10000

Average response latency in microseconds of the last 10000 packets.

6.15 latency-avg1000000

Average response latency in microseconds of the last 1000000 packets.

6.16 latency-slow

Number of queries answered in more than 1 second.

6.17 latency0-1

Number of queries answered in less than 1 ms.

6.18 latency1-10

Number of queries answered in 1-10 ms.

6.19 latency10-50

Number of queries answered in 10-50 ms.

6.20 latency50-100

Number of queries answered in 50-100 ms.

6.21 latency100-1000

Number of queries answered in 100-1000 ms.

6.22 no-policy

Number of queries dropped because no server was available.

6.23 noncompliant-queries

Number of queries dropped as non-compliant.

6.24 noncompliant-responses

Number of answers from a backend dropped as non-compliant.

6.25 queries

Number of received queries.

6.26 rdqueries

Number of received queries with the recursion desired bit set.

6.27 real-memory-usage

Current memory usage.

6.28 responses

Number of responses received from backends.

6.29 rule-drop

Number of queries dropped because of a rule.

6.30 rule-nxdomain

Number of NXDomain answers returned because of a rule.

6.31 rule-refused

Number of Refused answers returned because of a rule.

6.32 self-answered

Number of self-answered responses.

6.33 servfail-responses

Number of servfail answers received from backends.

6.34 trunc-failures

Number of errors encountered while truncating an answer.

6.35 uptime

Uptime of the dnsdist process, in seconds.

CHAPTER

SEVEN

CACHING RESPONSES

dnsdist implements a simple but effective packet cache, not enabled by default. It is enabled per-pool, but the same cache can be shared between several pools. The first step is to define a cache with *newPacketCache()*, then to assign that cache to the chosen pool, the default one being represented by the empty string:

```
pc = newPacketCache(10000, 86400, 0, 60, 60, false)
getPool(""):setCache(pc)
```

The first parameter (10000) is the maximum number of entries stored in the cache, and is the only one required. All the other parameters are optional and in seconds, except the last one which is a boolean. The second one (86400) is the maximum lifetime of an entry in the cache, the third one (0) is the minimum TTL an entry should have to be considered for insertion in the cache, the fourth one (60) is the TTL used for a Server Failure or a Refused response. The fifth one (60) is the TTL that will be used when a stale cache entry is returned. The sixth one is a boolean that when set to true, avoids reducing the TTL of cached entries.

For performance reasons the cache will pre-allocate buckets based on the maximum number of entries, so be careful to set the first parameter to a reasonable value. Something along the lines of a dozen bytes per pre-allocated entry can be expected on 64-bit. That does not mean that the memory is completely allocated up-front, the final memory usage depending mostly on the size of cached responses and therefore varying during the cache's lifetime. Assuming an average response size of 512 bytes, a cache size of 10000000 entries on a 64-bit host with 8GB of dedicated RAM would be a safe choice.

The *setStaleCacheEntriesTTL()* directive can be used to allow dnsdist to use expired entries from the cache when no backend is available. Only entries that have expired for less than n seconds will be used, and the returned TTL can be set when creating a new cache with *newPacketCache()*.

A reference to the cache affected to a specific pool can be retrieved with:

```
getPool("poolname"):getCache()
```

And removed with:

```
getPool("poolname"):unsetCache()
```

Cache usage stats (hits, misses, deferred inserts and lookups, collisions) can be displayed by using the *PacketCache:printStats()* method:

```
getPool("poolname"):getCache():printStats()
```

Expired cached entries can be removed from a cache using the *PacketCache:purgeExpired()* method, which will remove expired entries from the cache until at most n entries remain in the cache. For example, to remove all expired entries:

```
getPool("poolname"):getCache():purgeExpired(0)
```

Specific entries can also be removed using the *PacketCache:expungeByName()* method:

```
getPool("poolname"):getCache():expungeByName(newDNSName("powerdns.com"), dnsdist.A)
```

Finally, the *PacketCache:expunge()* method will remove all entries until at most n entries remain in the cache:

```
getPool("poolname"):getCache():expunge(0)
```

CHAPTER

EIGHT

EXPORTING STATISTICS VIA CARBON

8.1 Setting up a carbon export

To emit metrics to Graphite, or any other software supporting the Carbon protocol, use:

```
carbonServer('ip-address-of-carbon-server', 'ourname', 30)
```

Where `ourname` can be used to override your hostname, and `30` is the reporting interval in seconds. The last two arguments can be omitted. The latest version of PowerDNS Metronome comes with attractive graphs for dnsdist by default.

8.2 Query counters

In addition to other metrics, it is possible to send per-records statistics of the amount of queries by using `setQueryCount()`. With query counting enabled, dnsdist will increase a counter for every unique record or the behaviour you define in a custom Lua function by setting `setQueryCountFilter()`. This filter can decide whether to keep count on a query at all or rewrite for which query the counter will be increased. An example of a QueryCountFilter would be:

```
function filter(dq)
  qname = dq.qname:toString()

  -- don't count PTRs at all
  if(qname:match('in%-addr.arpa$')) then
    return false, ""
  end

  -- count these queries as if they were queried without leading www.
  if(qname:match('^www.')) then
    qname = qname:gsub('^www.', '')
  end

  -- count queries by default
  return true, qname
end

setQueryCountFilter(filter)
```

Valid return values for `QueryCountFilter` functions are:

- true: count the specified query
- false: don't count the query

Note that the query counters are buffered and flushed each time statistics are sent to the carbon server. The current content of the buffer can be inspected with :`getQueryCounters()`. If you decide to enable query counting

without *carbonServer()*, make sure you implement clearing the log from `maintenance()` by issuing `clearQueryCounters()`.

CHAPTER

NINE

WORKING WITH THE DNSDIST CONSOLE

dnsdist can expose a commandline console over an encrypted tcp connection for controlling it, debugging DNS issues and retrieving statistics.

The console can be enabled with *controlSocket()*:

```
controlSocket('192.0.2.53:5199')
```

To enable encryption, first generate a key with *makeKey()*:

```
$ ./dnsdist -l 127.0.0.1:5300
[..]
> makeKey()
setKey("ENCODED KEY")
```

Add the generated *setKey()* line to you dnsdist configuration file, along with a *controlSocket()*:

```
controlSocket('192.0.2.53:5199') -- Listen on this IP and port for client␣
↪connections
setKey("ENCODED KEY")            -- Shared secret for the console
```

Now you can run `dnsdist -c` to connect to the console. This makes dnsdist read its configuration file and use the *controlSocket()* and *setKey()* statements to set up its connection to the server.

If you want to connect over the network, create a configuration file with the same two statements and run `dnsdist -C /path/to/configfile -c`.

Alternatively, you can specify the address and key on the client commandline:

```
dnsdist -k "ENCODED KEY" -c 192.0.2.53:5199
```

Warning: This will leak the key into your shell's history and is **not** recommended.

CHAPTER

TEN

DNSCRYPT

dnsdist, when compiled with `--enable-dnscrypt`, can be used as a DNSCrypt server, uncurving queries before forwarding them to downstream servers and curving responses back. To make **dnsdist** listen to incoming DNSCrypt queries on 127.0.0.1 port 8443, with a provider name of "2.providername", using a resolver certificate and associated key stored respectively in the resolver.cert and resolver.key files, the *addDNSCryptBind()* directive can be used:

```
addDNSCryptBind("127.0.0.1:8443", "2.providername", "/path/to/resolver.cert", "/
↪path/to/resolver.key")
```

To generate the provider and resolver certificates and keys, you can simply do:

```
> generateDNSCryptProviderKeys("/path/to/providerPublic.key", "/path/to/
↪providerPrivate.key")
Provider fingerprint is: ␣
↪E1D7:2108:9A59:BF8D:F101:16FA:ED5E:EA6A:9F6C:C78F:7F91:AF6B:027E:62F4:69C3:B1AA
> generateDNSCryptCertificate("/path/to/providerPrivate.key", "/path/to/resolver.
↪cert", "/path/to/resolver.key", serial, validFrom, validUntil)
```

Ideally, the certificates and keys should be generated on an offline dedicated hardware and not on the resolver. The resolver key should be regularly rotated and should never touch persistent storage, being stored in a tmpfs with no swap configured.

You can display the currently configured DNSCrypt binds with:

```
> showDNSCryptBinds()
#   Address              Provider Name        Serial   Validity              P. ␣
↪Serial P. Validity
0   127.0.0.1:8443       2.name               14       2016-04-10 08:14:15   0    ␣
↪    -
```

If you forgot to write down the provider fingerprint value after generating the provider keys, you can use *printDNSCryptProviderFingerprint()* to retrieve it later:

```
> printDNSCryptProviderFingerprint("/path/to/providerPublic.key")
Provider fingerprint is: ␣
↪E1D7:2108:9A59:BF8D:F101:16FA:ED5E:EA6A:9F6C:C78F:7F91:AF6B:027E:62F4:69C3:B1A
```

CHAPTER

ELEVEN

CONFIGURING DOWNSTREAM SERVERS

As dnsdist is a loadbalancer and does not do any DNS resolving or serving by itself, it needs downstream servers. To add downstream servers, either include them on the command line:

```
dnsdist -l 130.161.252.29 -a 130.161.0.0/16 8.8.8.8 208.67.222.222 2620:0:ccc::2␣
↪2620:0:ccd::2
```

Or add them to the configuration file:

```
setLocal("130.161.252.29:53")
setACL("130.161.0.0/16")
newServer("8.8.8.8")
newServer("208.67.222.222")
newServer("2620:0:ccc::2")
newServer("2620:0:0ccd::2")
```

These two equivalent configurations give you sane load balancing using a very sensible distribution policy. Many users will simply be done with this configuration. It works as well for authoritative as for recursive servers.

11.1 Healthcheck

dnsdist uses a health check, sent once every second, to determine the availability of a backend server.

By default, an A query for "a.root-servers.net." is sent. A different query type, class and target can be specified by passing, respectively, the `checkType`, `checkClass` and `checkName` parameters to *`newServer()`*.

The default behavior is to consider any valid response with an RCODE different from ServFail as valid. If the `mustResolve` parameter of *`newServer()`* is set to `true`, a response will only be considered valid if its RCODE differs from NXDomain, ServFail and Refused.

The number of health check failures before a server is considered down is configurable via the `maxCheckFailures` parameter, defaulting to 1. The CD flag can be set on the query by setting `setCD` to true. e.g.:

```
newServer({address="192.0.2.1", checkType="AAAA", checkType=DNSClass.CHAOS,␣
↪checkName="a.root-servers.net.", mustResolve=true})
```

11.2 Source address selection

In multi-homed setups, it can be useful to be able to select the source address or the outgoing interface used by dnsdist to contact a downstream server. This can be done by using the *source* parameter:

```
newServer({address="192.0.2.1", source="192.0.2.127"})
newServer({address="192.0.2.1", source="eth1"})
newServer({address="192.0.2.1", source="192.0.2.127@eth1"})
```

The supported values for source are: - an IPv4 or IPv6 address, which must exist on the system - an interface name - an IPv4 or IPv6 address followed by '@' then an interface name

Please note that specifying the interface name is only supported on system having *IP_PKTINFO*.

CHAPTER

TWELVE

DYNAMIC RULE GENERATION

To set dynamic rules, based on recent traffic, define a function called *maintenance()* in Lua. It will get called every second, and from this function you can set rules to block traffic based on statistics. More exactly, the thread handling the *maintenance()* function will sleep for one second between each invocation, so if the function takes several seconds to complete it will not be invoked exactly every second.

As an example:

```
function maintenance()
    addDynBlocks(exceedQRate(20, 10), "Exceeded query rate", 60)
end
```

This will dynamically block all hosts that exceeded 20 queries/s as measured over the past 10 seconds, and the dynamic block will last for 60 seconds.

Dynamic blocks in force are displayed with *showDynBlocks()* and can be cleared with *clearDynBlocks()*. They return a table whose key is a *ComboAddress* object, representing the client's source address, and whose value is an integer representing the number of queries matching the corresponding condition (for example the qtype for *exceedQTypeRate()*, rcode for *exceedServFails()*).

All exceed-functions are documented in the *Configuration Reference*.

Dynamic blocks drop matched queries by default, but this behavior can be changed with *setDynBlocksAction()*. For example, to send a REFUSED code instead of droppping the query:

```
setDynBlocksAction(DNSAction.Refused)
```

CHAPTER

THIRTEEN

GUIDES

These chapters contain several guides and nuggets of information regarding dnsdist operation and accomplishing specific goals.

13.1 Built-in webserver

To visually interact with dnsdist, try add `webserver()` to the configuration:

```
webserver("127.0.0.1:8083", "supersecretpassword", "supersecretAPIkey")
```

Now point your browser at http://127.0.0.1:8083 and log in with any username, and that password. Enjoy!

13.1.1 Security of the Webserver

The built-in webserver serves its content from inside the binary, this means it will not and connot read from disk.

By default, our web server sends some security-related headers:

```
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-Permitted-Cross-Domain-Policies: none
X-XSS-Protection: 1; mode=block
Content-Security-Policy: default-src 'self'; style-src 'self' 'unsafe-inline'
```

You can override those headers, or add custom headers by using the last parameter to `webserver()`. For example, to remove the X-Frame-Options header and add a X-Custom one:

```
webserver("127.0.0.1:8080", "supersecret", "apikey", {["X-Frame-Options"]= "", ["X-
↪Custom"]="custom"}
```

13.1.2 dnsdist API

To access the API, the *apikey* must be set in the `webserver()` function. Use the API, this key will need to be sent to dnsdist in the `X-API-Key` request header. An HTTP 401 response is returned when a wrong or no API key is received. A 404 response is generated is the requested endpoint does not exist. And a 405 response is returned when the HTTP methos is not allowed.

URL Endpoints

`GET /jsonstat`
Get statistics from dnsdist in JSON format. The `Accept` request header is ignored. This endpoint accepts a `command` query for different statistics:

- `stats`: Get all *Statistics* as a JSON dict

- `dynblocklist`: Get all current *dynamic blocks*, keyed by netmask
- `ebpfblocklist`: Idem, but for *eBPF* blocks

Example request:

```
GET /jsonstat?command=stats HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Security-Policy: default-src 'self'; style-src 'self' 'unsafe-
↪inline'
Content-Type: application/json
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-Permitted-Cross-Domain-Policies: none
X-Xss-Protection: 1; mode=block

{"acl-drops": 0, "block-filter": 0, "cache-hits": 0, "cache-misses": 0,
↪"cpu-sys-msec": 633, "cpu-user-msec": 499, "downstream-send-errors": 0,
↪"downstream-timeouts": 0, "dyn-block-nmg-size": 1, "dyn-blocked": 3,
↪"empty-queries": 0, "fd-usage": 17, "latency-avg100": 7651.3982737482893,
↪ "latency-avg1000": 860.05142763680249, "latency-avg10000": 87.
↪032142373878372, "latency-avg1000000": 0.87146026426551759, "latency-slow
↪": 0, "latency0-1": 0, "latency1-10": 0, "latency10-50": 22, "latency100-
↪1000": 1, "latency50-100": 0, "no-policy": 0, "noncompliant-queries": 0,
↪"noncompliant-responses": 0, "over-capacity-drops": 0, "packetcache-hits
↪": 0, "packetcache-misses": 0, "queries": 26, "rdqueries": 26, "real-
↪memory-usage": 6078464, "responses": 23, "rule-drop": 0, "rule-nxdomain
↪": 0, "rule-refused": 0, "self-answered": 0, "server-policy":
↪"leastOutstanding", "servfail-responses": 0, "too-old-drops": 0, "trunc-
↪failures": 0, "uptime": 412}
```

Example request:

```
GET /jsonstat?command=dynblocklist HTTP/1.1
Host: example.com
Accept: application/json, text/javascript
```

Example response:

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Security-Policy: default-src 'self'; style-src 'self' 'unsafe-
↪inline'
Content-Type: application/json
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-Permitted-Cross-Domain-Policies: none
X-Xss-Protection: 1; mode=block

{"127.0.0.1/32": {"blocks": 3, "reason": "Exceeded query rate", "seconds":␣
↪10}}
```

Query Parameters

- **command** – one of `stats`, `dynblocklist` or `ebpfblocklist`

GET /api/v1/servers/localhost
Get a quick overview of several parameters.

Response JSON Object

- **acl** (*string*) – A string of comma-separated netmasks currently allowed by the *ACL*.
- **cache-hit-response-rules** (*list*) – A list of *ResponseRule* objects applied on cache hits
- **daemon_type** (*string*) – The type of daemon, always "dnsdist"
- **frontends** (*list*) – A list of *Frontend* objects
- **pools** (*list*) – A list of *Pool* objects
- **response-rules** (*list*) – A list of *ResponseRule* objects
- **rules** (*list*) – A list of *Rule* objects
- **servers** (*list*) – A list of *Server* objects
- **version** (*string*) – The running version of dnsdist

GET /api/v1/servers/localhost/statistics
Returns a list of all statistics as *StatisticItem*.

GET /api/v1/servers/localhost/config
Returns a list of *ConfigSetting* objects.

GET /api/v1/servers/localhost/config/allow-from
Gets you the `allow-from` *ConfigSetting*, who's value is a list of strings of all the netmasks in the *ACL*.

PUT /api/v1/servers/localhost/config/allow-from
Allows you to add to the ACL. TODO **how**

JSON Objects

ConfigSetting
An object representing a global configuration element. The following configuration are returned:

- `acl` The currently configured *ACLs*
- `control-socket` The currently configured *console address*
- `ecs-override`
- `ecs-source-prefix-v4` The currently configured *setECSSourcePrefixV4()*
- `ecs-source-prefix-v6` The currently configured *setECSSourcePrefixV6()*
- `fixup-case`
- `max-outstanding`
- `server-policy` The currently set *Loadbalancing and Server Policies*
- `stale-cache-entries-ttl`
- `tcp-recv-timeout`
- `tcp-send-timeout`
- `truncate-tc`
- `verbose`
- `verbose-health-checks`

Object Properties

- **name** (*string*) – The name of the setting
- **type** (*string*) – "ConfigSetting"
- **value** (*string*) – The value for this setting

Frontend
A description of a bind dnsdist is listening on.

Object Properties

- **address** (*string*) – IP and port that is listened on
- **id** (*integer*) – Internal identifier
- **queries** (*integer*) – The number of received queries on this bind
- **udp** (*boolean*) – true if this is a UDP bind
- **tcp** (*boolean*) – true if this is a TCP bind

Pool
A description of a pool of backend servers.

Object Properties

- **id** (*integer*) – Internal identifier
- **cacheDeferredInserts** (*integer*) – The number of times an entry could not be inserted in the associated cache, if any, because of a lock
- **cacheDeferredLookups** (*integer*) – The number of times an entry could not be looked up from the associated cache, if any, because of a lock
- **cacheEntries** (*integer*) – The current number of entries in the associated cache, if any
- **cacheHits** (*integer*) – The number of cache hits for the associated cache, if any
- **cacheLookupCollisions** (*integer*) – The number of times an entry retrieved from the cache based on the query hash did not match the actual query
- **cacheInsertCollisions** (*integer*) – The number of times an entry could not be inserted into the cache because a different entry with the same hash already existed
- **cacheMisses** (*integer*) – The number of cache misses for the associated cache, if any
- **cacheSize** (*integer*) – The maximum number of entries in the associated cache, if any
- **cacheTTLTooShorts** (*integer*) – The number of times an entry could not be inserted into the cache because its TTL was set below the minimum threshold
- **name** (*string*) – Name of the pool
- **serversCount** (*integer*) – Number of backends in this pool

Rule
This represents a policy that is applied to queries

Object Properties

- **action** (*string*) – The action taken when the rule matches (e.g. "to pool abuse")
- **action-stats** (*dict*) – A list of statistics whose content varies depending on the kind of rule
- **id** (*integer*) – The identifier (or order) of this rule
- **matches** (*integer*) – How many times this rule was hit

- **rule** (*string*) – The matchers for the packet (e.g. "qname==bad-domain1.example., bad-domain2.example.")

ResponseRule
This represents a policy that is applied to responses

Object Properties

- **action** (*string*) – The action taken when the rule matches (e.g. "drop")
- **id** (*integer*) – The identifier (or order) of this rule
- **matches** (*integer*) – How many times this rule was hit
- **rule** (*string*) – The matchers for the packet (e.g. "qname==bad-domain1.example., bad-domain2.example.")

Server
This object represents a backend server.

Object Properties

- **address** (*string*) – The remote IP and port
- **id** (*integer*) – Internal identifier
- **latency** (*integer*) – The current latency of this backend server
- **name** (*string*) – The name of this server
- **order** (*integer*) – Order number
- **outstanding** (*integer*) – Number of currently outstanding queries
- **pools** (*[string]*) – The pools this server belongs to
- **qps** (*integer*) – The current number of queries per second to this server
- **qpsLimit** (*integer*) – The configured maximum number of queries per second
- **queries** (*integer*) – Total number of queries sent to this backend
- **reuseds** (*integer*) – Number of queries for which a response was not received in time
- **sendErrors** (*integer*) – Number of network errors while sending a query to this server
- **state** (*string*) – The state of the server (e.g. "DOWN" or "up")
- **weight** (*integer*) – The weight assigned to this server

StatisticItem
This represents a statistics element.

Object Properties

- **name** (*string*) – The name of this statistic. See *Statistics*
- **type** (*string*) – "StatisticItem"
- **value** (*integer*) – The value for this item

13.2 Server pools

dnsdist has the concept to "server pools", any number of servers can belong to a group.

Let's say we know we're getting a whole bunch of traffic for a domain used in DoS attacks, for example 'example.com'. We can do two things with this kind of traffic. Either we block it outright, like this:

```
addAction("bad-domain.example.", dropAction())
```

Or we configure a server pool dedicated to receiving the nasty stuff:

```
newServer({address="192.0.2.3", pool="abuse"})          -- Add a backend server␣
↪with address 192.0.2.3 and assign it to the "abuse" pool
addAction({'bad-domain1.example', 'bad-domain2.example.'}, PoolAction("abuse")) --␣
↪Send all queries for "bad-domain1.example." and "bad-domain2.example" to the
↪"abuse" pool
```

The wonderful thing about this last solution is that it can also be used for things where a domain might possibly be legit, but it is still causing load on the system and slowing down the internet for everyone. With such an abuse server, 'bad traffic' still gets a chance of an answer, but without impacting the rest of the world (too much).

We can similarly add clients to the abuse server:

```
addAction({"192.168.12.0/24", "192.168.13.14"}, PoolAction("abuse"))
```

To define a pool that should receive only a *QPS*-limited amount of traffic, do:

```
addAction("com.", QPSPoolAction(10000, "gtld-cluster"))
```

Traffic exceeding the *QPS* limit will not match that rule, and subsequent rules will apply normally.

Servers can be added to or removed from pools with the *Server:addPool()* and *Server:rmPool()* functions respectively:

```
getServer(4):addPool("abuse")
getServer(4):rmPool("abuse")
```

13.3 Loadbalancing and Server Policies

dnsdist selects the server (if there are multiple eligable) to send queries to based on the configured policy. Only servers that are marked as 'up', either forced so by the administrator or as the result of the last health check, might be selected.

13.3.1 Built-in Policies

leastOutstanding

The default load balancing policy is called `leastOutstanding`, which means the server with the least queries 'in the air' is picked. The exact selection algorithm is:

- pick the server with the least queries 'in the air' ;
- in case of a tie, pick the one with the lowest configured 'order' ;
- in case of a tie, pick the one with the lowest measured latency (over an average on the last 128 queries answered by that server).

firstAvailable

The `firstAvailable` policy, picks the first available server that has not exceeded its QPS limit, ordered by increasing 'order'. If all servers are above their QPS limit, a server is selected based on the `leastOutstanding` policy. For now this is the only policy using the QPS limit.

wrandom

A further policy, `wrandom` assigns queries randomly, but based on the weight parameter passed to *`newServer()`*.

For example, if two servers are available, the first one with a weigth of 2 and the second one with a weight of 1 (the default), the first one should get two thirds of the incoming queries and the second one the remaining third.

whashed

`whashed` is a similar weighted policy, but assigns questions with identical hash to identical servers, allowing for better cache concentration ('sticky queries'). The current hash algorithm is based on the qname of the query.

setWHashedPertubation(*value*)
Set the hash perturbation value to be used in the whashed policy instead of a random one, allowing to have consistent whashed results on different instances.

roundrobin

The last available policy is `roundrobin`, which indiscriminately sends each query to the next server that is up.

13.3.2 Lua server policies

If you don't like the default policies you can create your own, like this for example:

```
counter=0
function luaroundrobin(servers, dq)
     counter=counter+1
     return servers[1+(counter % #servers)]
end

setServerPolicyLua("luaroundrobin", luaroundrobin)
```

Incidentally, this is similar to setting: `setServerPolicy(roundrobin)` which uses the C++ based roundrobin policy.

Or:

```
newServer("192.168.1.2")
newServer({address="8.8.4.4", pool="numbered"})

function splitSetup(servers, dq)
  if(string.match(dq.qname:toString(), "%d"))
  then
    print("numbered pool")
    return leastOutstanding.policy(getPoolServers("numbered"), dq)
  else
    print("standard pool")
    return leastOutstanding.policy(servers, dq)
  end
end

setServerPolicyLua("splitsetup", splitSetup)
```

13.3.3 ServerPolicy Objects

class ServerPolicy
This represents a server policy. The built-in policies are of this type

ServerPolicy.**policy**(*servers*, *dq*) → Server
Run the policy to receive the server it has selected.

Parameters

- **servers** – A list of *Server* objects
- **dq** (DNSQuestion) – The incoming query

13.3.4 Functions

newServerPolicy(*name*, *function*) → ServerPolicy
Create a policy object from a Lua function. function must match the prototype for *ServerPolicy.policy()*.

Parameters

- **name** (*string*) – Name of the policy
- **function** (*string*) – The function to call for this policy

setServerPolicy(*policy*)
Set server selection policy to policy.

Parameters **policy** (ServerPolicy) – The policy to use

setServerPolicyLua(*name*, *function*)
Set server selection policy to one named *name'* and provided by function.

Parameters

- **name** (*string*) – name for this policy
- **function** (*string*) – name of the function

setServFailWhenNoServer(*value*)
If set, return a ServFail when no servers are available, instead of the default behaviour of dropping the query.

Parameters **value** (*bool*) –

setPoolServerPolicy(*policy*, *pool*)
Set the server selection policy for pool to policy.

Parameters

- **policy** (ServerPolicy) – The policy to apply
- **pool** (*string*) – Name of the pool

setPoolServerPolicyLua(*name*, *function*, *pool*)
Set the server selection policy for pool to one named name and provided by function.

Parameters

- **name** (*string*) – name for this policy
- **function** (*string*) – name of the function
- **pool** (*string*) – Name of the pool

showPoolServerPolicy(*pool*)
Print server selection policy for pool.

Parameters **pool** (*string*) – The pool to print the policy for

CHAPTER

FOURTEEN

ADVANCED TOPICS

These chapters contain information on the advanced features of dnsdist

14.1 Access Control

dnsdist can be used to front traditional recursive nameservers, these usually come with a way to limit the network ranges that may query it to prevent becoming an *open resolver*. To be a good internet citizen, dnsdist by default listens on the loopback address (*127.0.0.1:53*) and limits queries to these loopback, **RFC 1918** and other local addresses:

- `127.0.0.0/8`
- `10.0.0.0/8`
- `100.64.0.0/10`
- `169.254.0.0/16`
- `192.168.0.0/16`
- `172.16.0.0/12`
- `::1/128`
- `fc00::/7`
- `fe80::/10`

Further more, dnsdist only listens for queries on the local-loopback interface by default.

14.1.1 Listening on different addresses

To listen on other addresses than just the local addresses, use *`setLocal()`* and *`addLocal()`*.

`setLocal()` **resets** the list of current listen addresses to the specified address and *`addLocal()`* adds an additional listen address. To listen on `127.0.0.1:5300`, `192.0.2.1:53` and UDP-only on `[2001:db8::15::47]:53`, configure the following:

```
setLocal('127.0.0.1:5300')
addLocal('192.0.2.1') -- Port 53 is default is none is specified
addLocal('2001:db8::15::47', false)
```

Listen addresses cannot be modified at runtime and must be specified in the configuration file.

As dnsdist is IPv4 and IPv6 agnostic, this means that dnsdist internally does not know the difference. So feel free to listen on the magic `0.0.0.0` or `::` addresses, dnsdist does the right thing to set the return address of queries, but set your *ACL* properly.

14.1.2 Modifying the ACL

ACLs can be modfied at runtime from the *Working with the dnsdist Console*. To inspect the currently active *ACL*, run *showACL()*.

To add a new network range to the existing ACL, use *addACL()*:

```
addACL('192.0.2.0/25')
addACL('2001:db8::1') -- No netmask specified, only allow this address
```

dnsdist also has the *setACL()* function that accepts a list of netmasks and resets the ACL to that list:

```
setACL({'192.0.2.0/25', '2001:db8:15::bea/64'})
```

14.2 TeeAction: copy the DNS traffic stream

This action sends off a copy of a UDP query to another server, and keeps statistics on the responses received. Sample use:

```
> addAction(AllRule(), TeeAction("192.0.2.54"))
> getAction(0):printStats()
refuseds     0
nxdomains    0
noerrors     0
servfails    0
recv-errors  0
tcp-drops    0
responses    0
other-rcode  0
send-errors  0
queries 0
```

It is also possible to share a *TeeAction()* between several rules. Statistics will be combined in that case.

14.3 Lua actions in rules

While we can pass every packet through the `blockFilter()` functions, it is also possible to configure **dnsdist** to only hand off some packets for Lua inspection. If you think Lua is too slow for your query load, or if you are doing heavy processing in Lua, this may make sense.

To select specific packets for Lua attention, use *addLuaAction()* or `addLuaReponseAction()`.

A sample configuration could look like this:

```
function luarule(dq)
  if(dq.qtype==35) -- NAPTR
  then
    return DNSAction.Pool, "abuse" -- send to abuse pool
  else
    return DNSAction.None, ""      -- no action
  end
end

addLuaAction(AllRule(), luarule)
```

14.4 Runtime-modifiable IP address sets

New in version 1.2.0.

From within *maintenance()* or other places, we may find that certain IP addresses must be treated differently for a certain time.

This may be used to temporarily shunt traffic to another pool for example.

TimedIPSetRule() creates an object to which native IP addresses can be added in *ComboAddress* form.

TimedIPSetRule() → TimedIPSetRule
Returns a *TimedIPSetRule*.

class TimedIPSetRule
Can be used to handle IP addresses differentlt based on the date and time

classmethod TimedIPSetRule:**add**(*address*, *seconds*)
Add an IP address to the set for the next `second` seconds.

Parameters

- **address** (ComboAddress) – The address to add
- **seconds** (*int*) – Time to keep the address in the Rule

classmethod TimedIPSetRule:**cleanup**()
Purge the set from expired IP addresses

classmethod TimedIPSetRule:**clear**()
Clear the entire set

classmethod TimedIPSetRule:**slice**()
Convert the TimedIPSetRule into a DNSRule that can be passed to *addAction()*

A working example:

```
tisrElGoog=TimedIPSetRule()
tisrRest=TimedIPSetRule()
addAction(tisrElGoog:slice(), PoolAction("elgoog"))
addAction(tisrRest:slice(), PoolAction(""))

elgoogPeople=newNMG()
elgoogPeople:addMask("192.168.5.0/28")

function pickPool(dq)
        if(elgoogPeople:match(dq.remoteaddr)) -- in real life, this would be␣
↪external
        then
                print("Lua caught query for a googlePerson")
                tisrElGoog:add(dq.remoteaddr, 10)
                return DNSAction.Pool, "elgoog"
        else
                print("Lua caught query for restPerson")
                tisrRest:add(dq.remoteaddr, 60)
                return DNSAction.None, ""
        end
end

addLuaAction(AllRule(), pickPool)
```

14.5 Using EDNS Client Subnet

In order to provide the downstream server with the address of the real client, or at least the one talking to dnsdist, the `useClientSubnet` parameter can be used when creating a *new server*. This parameter indicates whether an EDNS Client Subnet option should be added to the request. If the incoming request already contains an EDNS Client Subnet value, it will not be overriden unless `setECSOverride()` is set to `true`. The default source prefix-length is 24 for IPv4 and 56 for IPv6, meaning that for a query received from 192.0.2.42, the EDNS Client Subnet value sent to the backend will be 192.0.2.0. This can be changed with *setECSSourcePrefixV4()* and *setECSSourcePrefixV6()*.

In addition to the global settings, rules and Lua bindings can alter this behavior per query:

- calling *DisableECSAction()* or setting `dq.useECS` to `false` prevents the sending of the ECS option.
- calling *ECSOverrideAction()* or setting `dq.ecsOverride` will override the global `setECSOverride()` value.
- calling `ECSPrefixLengthAction(v4, v6)()` or setting `dq.ecsPrefixLength` will override the global *setECSSourcePrefixV4()* and *setECSSourcePrefixV6()* values.

In effect this means that for the EDNS Client Subnet option to be added to the request, `useClientSubnet` should be set to `true` for the backend used (default to `false`) and ECS should not have been disabled by calling *DisableECSAction()* or setting `dq.useECS` to `false` (default to true).

14.6 Rules for traffic exceeding QPS limits

Traffic that exceeds a QPS limit, in total or per IP (subnet) can be matched by the *MaxQPSIPRule()*-rule. For example:

```
addAction(MaxQPSIPRule(5, 32, 48), DelayAction(100))
```

This measures traffic per IPv4 address and per /48 of IPv6, and if traffic for such an address (range) exceeds 5 *qps*, it gets delayed by 100ms.

As another example:

```
addAction(MaxQPSIPRule(5), NoRecurseAction())
```

This strips the Recursion Desired (RD) bit from any traffic per IPv4 or IPv6 /64 that exceeds 5 qps. This means any those traffic bins is allowed to make a recursor do 'work' for only 5 qps.

If this is not enough, try:

```
addAction(MaxQPSIPRule(5), DropAction())
-- or
addAction(MaxQPSIPRule(5), TCAction())
```

This will respectively drop traffic exceeding that 5 QPS limit per IP or range, or return it with TC=1, forcing clients to fall back to TCP.

To turn this per IP or range limit into a global limit, use `NotRule(MaxQPSRule(5000))` instead of *MaxQPSIPRule()*.

14.7 eBPF Socket Filtering

dnsdist can use eBPF socket filtering on recent Linux kernels (4.1+) built with eBPF support (`CONFIG_BPF`, `CONFIG_BPF_SYSCALL`, ideally `CONFIG_BPF_JIT`). This feature might require an increase of the memory limit associated to a socket, via the sysctl setting `net.core.optmem_max`. When attaching an eBPF program

to a socket, the size of the program is checked against this limit, and the default value might not be enough. Large map sizes might also require an increase of `RLIMIT_MEMLOCK`.

This feature allows dnsdist to ask the kernel to discard incoming packets in kernel-space instead of them being copied to userspace just to be dropped, thus being a lot of faster.

The BPF filter can be used to block incoming queries manually:

```
> bpf = newBPFFilter(1024, 1024, 1024)
> bpf:attachToAllBinds()
> bpf:block(newCA("2001:DB8::42"))
> bpf:blockQName(newDNSName("evildomain.com"), 255)
> bpf:getStats()
[2001:DB8::42]: 0
evildomain.com. 255: 0
> bpf:unblock(newCA("2001:DB8::42"))
> bpf:unblockQName(newDNSName("evildomain.com"), 255)
> bpf:getStats()
```

The *BPFFilter:blockQName()* method can be used to block queries based on the exact qname supplied, in a case-insensitive way, and an optional qtype. Using the 255 (ANY) qtype will block all queries for the qname, regardless of the qtype. Contrary to source address filtering, qname filtering only works over UDP. TCP qname filtering can be done the usual way:

```
addAction(AndRule({TCPRule(true), makeRule("evildomain.com")}), DropAction())
```

The *BPFFilter:attachToAllBinds()* method attaches the filter to every existing bind at runtime, but it's also possible to define a default BPF filter at configuration time, so it's automatically attached to every bind:

```
bpf = newBPFFilter(1024, 1024, 1024)
setDefaultBPFFilter(bpf)
```

Finally, it's also possible to attach it to specific binds at runtime:

```
> bpf = newBPFFilter(1024, 1024, 1024)
> showBinds()
#   Address              Protocol  Queries
0   [::]:53              UDP       0
1   [::]:53              TCP       0
> bd = getBind(0)
> bd:attachFilter(bpf)
```

dnsdist also supports adding dynamic, expiring blocks to a BPF filter:

```
bpf = newBPFFilter(1024, 1024, 1024)
setDefaultBPFFilter(bpf)
dbpf = newDynBPFFilter(bpf)
function maintenance()
        addBPFFilterDynBlocks(exceedQRate(20, 10), dbpf, 60)
        dbpf:purgeExpired()
end
```

This will dynamically block all hosts that exceeded 20 queries/s as measured over the past 10 seconds, and the dynamic block will last for 60 seconds.

The dynamic eBPF blocks and the number of queries they blocked can be seen in the web interface and retrieved from the API. Note however that eBPF dynamic objects need to be registered before they appear in the web interface or the API, using the *registerDynBPFFilter()* function:

```
registerDynBPFFilter(dbpf)
```

They can be unregistered at a later point using the *unregisterDynBPFFilter()* function.

This feature has been successfully tested on Arch Linux, Arch Linux ARM, Fedora Core 23 and Ubuntu Xenial

14.8 Performance Tuning

First, a few words about **dnsdist** architecture:

- Each local bind has its own thread listening for incoming UDP queries
- and its own thread listening for incoming TCP connections, dispatching them right away to a pool of threads
- Each backend has its own thread listening for UDP responses
- A maintenance thread calls the maintenance() Lua function every second if any, and is responsible for cleaning the cache
- A health check thread checks the backends availability
- A control thread handles console connections
- A carbon thread exports statistics to carbon servers if needed
- One or more webserver threads handle queries to the internal webserver

The maximum number of threads in the TCP pool is controlled by the *setMaxTCPClientThreads()* directive, and defaults to 10. This number can be increased to handle a large number of simultaneous TCP connections. If all the TCP threads are busy, new TCP connections are queued while they wait to be picked up.

The maximum number of queued connections can be configured with *setMaxTCPQueuedConnections()* and defaults to 1000. Any value larger than 0 will cause new connections to be dropped if there are already too many queued. By default, every TCP worker thread has its own queue, and the incoming TCP connections are dispatched to TCP workers on a round-robin basis. This might cause issues if some connections are taking a very long time, since incoming ones will be waiting until the TCP worker they have been assigned to has finished handling its current query, while other TCP workers might be available.

The experimental *setTCPUseSinglePipe()* directive can be used so that all the incoming TCP connections are put into a single queue and handled by the first TCP worker available.

When dispatching UDP queries to backend servers, dnsdist keeps track of at most **n** outstanding queries for each backend. This number **n** can be tuned by the *setMaxUDPOutstanding()* directive, defaulting to 10240, with a maximum value of 65535. Large installations are advised to increase the default value at the cost of a slightly increased memory usage.

Most of the query processing is done in C++ for maximum performance, but some operations are executed in Lua for maximum flexibility:

- Rules added by *addLuaAction()*
- Server selection policies defined via *setServerPolicyLua()* or *newServerPolicy()*

While Lua is fast, its use should be restricted to the strict necessary in order to achieve maximum performance, it might be worth considering using LuaJIT instead of Lua. When Lua inspection is needed, the best course of action is to restrict the queries sent to Lua inspection by using *addLuaAction()* with a selector.

dnsdist design choices mean that the processing of UDP queries is done by only one thread per local bind. This is great to keep lock contention to a low level, but might not be optimal for setups using a lot of processing power, caused for example by a large number of complicated rules. To be able to use more CPU cores for UDP queries processing, it is possible to use the `reuseport` parameter of the *addLocal()* and *setLocal()* directives to be able to add several identical local binds to dnsdist:

```
addLocal("192.0.2.1:53", {reuseport=true})
addLocal("192.0.2.1:53", {reuseport=true})
addLocal("192.0.2.1:53", {reuseport=true})
addLocal("192.0.2.1:53", {reuseport=true})
```

dnsdist will then add four identical local binds as if they were different IPs or ports, start four threads to handle incoming queries and let the kernel load balance those randomly to the threads, thus using four CPU cores for rules processing. Note that this require `SO_REUSEPORT` support in the underlying operating system (added for example in Linux 3.9). Please also be aware that doing so will increase lock contention and might not therefore

scale linearly. This is especially true for Lua-intensive setups, because Lua processing in dnsdist is serialized by an unique lock for all threads.

Another possibility is to use the reuseport option to run several dnsdist processes in parallel on the same host, thus avoiding the lock contention issue at the cost of having to deal with the fact that the different processes will not share informations, like statistics or DDoS offenders.

The UDP threads handling the responses from the backends do not use a lot of CPU, but if needed it is also possible to add the same backend several times to the dnsdist configuration to distribute the load over several responder threads:

```
newServer({address="192.0.2.127:53", name="Backend1"})
newServer({address="192.0.2.127:53", name="Backend2"})
newServer({address="192.0.2.127:53", name="Backend3"})
newServer({address="192.0.2.127:53", name="Backend4"})
```

14.9 SNMP support

dnsdist supports exporting statistics and sending traps over SNMP when compiled with `Net SNMP` support, acting as an `AgentX` subagent. SNMP support is enabled via the *snmpAgent()* directive.

By default, the only traps sent when Traps are enabled, are backend status change notifications. But custom traps can also be sent:

- from Lua, with *sendCustomTrap()* and *DNSQuestion:sendTrap()*
- For selected queries and responses, using *SNMPTrapAction()* and *SNMPTrapResponseAction()*

`Net SNMP snmpd` doesn't accept subagent connections by default, so to use the SNMP features of **dnsdist** the following line should be added to the `snmpd.conf` configuration file:

```
master agentx
```

In addition to that, the permissions on the resulting socket might need to be adjusted so that the `dnsdist` user can write to it. This can be done with the following lines in `snmpd.conf` (assuming *dnsdist* is running as *dnsdist:dnsdist*):

```
agentxperms 0700 0700 dnsdist dnsdist
```

In order to allow the retrieval of statistics via SNMP, `snmpd`'s access control has to configured. A very simple SNMPv2c setup only needs the configuration of a read-only community in `snmpd.conf`:

```
rocommunity dnsdist42
```

`snmpd` also supports more secure SNMPv3 setup, using for example the `createUser` and `rouser` directives:

```
createUser myuser SHA "my auth key" AES "my enc key"
rouser myuser
```

`snmpd` can be instructed to send SNMPv2 traps to a remote SNMP trap receiver by adding the following directive to the `snmpd.conf` configuration file:

```
trap2sink 192.0.2.1
```

The description of **dnsdist**'s SNMP MIB is as follows:

```
-- -*- snmpv2 -*-
-- ---------------------------------------------------------------------
-- MIB file for dnsdist
-- ---------------------------------------------------------------------

```

```
DNSDIST-MIB DEFINITIONS ::= BEGIN

IMPORTS
    OBJECT-TYPE, MODULE-IDENTITY, enterprises,
    Counter64, Unsigned32, NOTIFICATION-TYPE
        FROM SNMPv2-SMI
    CounterBasedGauge64
        FROM HCNUM-TC
    Float64TC
        FROM FLOAT-TC-MIB
    OBJECT-GROUP, MODULE-COMPLIANCE, NOTIFICATION-GROUP
        FROM SNMPv2-CONF
    InetAddressType
        FROM INET-ADDRESS-MIB
    TEXTUAL-CONVENTION, DisplayString
        FROM SNMPv2-TC;

dnsdist MODULE-IDENTITY
    LAST-UPDATED "201611080000Z"
    ORGANIZATION "PowerDNS BV"
    CONTACT-INFO "support@powerdns.com"
    DESCRIPTION
       "This MIB module describes information gathered through dnsdist."

    REVISION "201611080000Z"
    DESCRIPTION "Initial revision."

    ::= { powerdns 3 }

powerdns                OBJECT IDENTIFIER ::= { enterprises 43315 }

stats OBJECT IDENTIFIER ::= { dnsdist 1 }

queries OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries received"
    ::= { stats 1 }

responses OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of responses received"
    ::= { stats 2 }

servfailResponses OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of servfail responses received"
    ::= { stats 3 }

aclDrops OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries dropped because of the ACL"
```

```
    ::= { stats 4 }

-- stats 5 was a BlockFilter Counter, removed in 1.2.0

ruleDrop OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries dropped because of a rule"
    ::= { stats 6 }

ruleNXDomain OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of NXDomain responses returned because of a rule"
    ::= { stats 7 }

ruleRefused OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of Refused responses returned because of a rule"
    ::= { stats 8 }

selfAnswered OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of self-answered responses"
    ::= { stats 9 }

downstreamTimeouts OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of downstream timeouts"
    ::= { stats 10 }

downstreamSendErrors OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of downstream send errors"
    ::= { stats 11 }

truncFailures OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of errors while truncating a response"
    ::= { stats 12 }

noPolicy OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
```

```
    STATUS current
    DESCRIPTION
        "Number of queries dropped because no server was available"
    ::= { stats 13 }

latency01 OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries answered in less than 1 ms"
    ::= { stats 14 }

latency110 OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries answered in 1-10 ms"
    ::= { stats 15 }

latency1050 OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries answered in 10-50 ms"
    ::= { stats 16 }

latency50100 OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries answered in 50-100 ms"
    ::= { stats 17 }

latency1001000 OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries answered in 100-1000 ms"
    ::= { stats 18 }

latencySlow OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries answered in more than 1s"
    ::= { stats 19 }

latencyAVG100 OBJECT-TYPE
    SYNTAX Float64TC
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Average latency over the last 100 queries"
    ::= { stats 20 }

latencyAVG1000 OBJECT-TYPE
    SYNTAX Float64TC
```

```
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Average latency over the last 1000 queries"
    ::= { stats 21 }

latencyAVG10000 OBJECT-TYPE
    SYNTAX Float64TC
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Average latency over the last 10000 queries"
    ::= { stats 22 }

latencyAVG1000000 OBJECT-TYPE
    SYNTAX Float64TC
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Average latency over the last 1000000 queries"
    ::= { stats 23 }

uptime OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Uptime of the dnsdist process, in seconds"
    ::= { stats 24 }

realMemoryUsage OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Memory usage"
    ::= { stats 25 }

nonCompliantQueries OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries dropped as non-compliant"
    ::= { stats 26 }

nonCompliantResponses OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of responses dropped as non-compliant"
    ::= { stats 27 }

rdQueries OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries with the RD flag set"
    ::= { stats 28 }

emptyQueries OBJECT-TYPE
```

```
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of empty queries received"
    ::= { stats 29 }

cacheHits OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of cache hits"
    ::= { stats 30 }

cacheMisses OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of cache misses"
    ::= { stats 31 }

cpuUserMSec OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "CPU Usage (user)"
    ::= { stats 32 }

cpuSysMSec OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "CPU Usage (sys)"
    ::= { stats 33 }

fdUsage OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of file descriptors"
    ::= { stats 34 }

dynBlocked OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries dropped because of a dynamic block"
    ::= { stats 35 }

dynBlockNMGSize OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Dynamic blocks (NMG) size"
    ::= { stats 36 }
```

```
backendStatTable OBJECT-TYPE
    SYNTAX SEQUENCE OF BackendStatEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION "Statistics for backends"
    ::= { dnsdist 2 }

backendStatEntry OBJECT-TYPE
    SYNTAX BackendStatEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION "Statistics for one backend"
    INDEX { backendId }
    ::= { backendStatTable 1 }

BackendStatEntry ::= SEQUENCE {
    backendId           Unsigned32,
    backendName         DisplayString,
    backendLatency      CounterBasedGauge64,
    backendWeight       CounterBasedGauge64,
    backendOutstanding  CounterBasedGauge64,
    backendQPSLimit     CounterBasedGauge64,
    backendReused       Counter64,
    backendState        DisplayString,
    backendAddress      OCTET STRING,
    backendPools        DisplayString,
    backendQPS          CounterBasedGauge64,
    backendQueries      Counter64,
    backendOrder        CounterBasedGauge64
}

backendId OBJECT-TYPE
    SYNTAX Unsigned32
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION
        "Backend ID"
    ::= { backendStatEntry 1 }

backendName OBJECT-TYPE
    SYNTAX DisplayString
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend name"
    ::= { backendStatEntry 2 }

backendLatency OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend latency"
    ::= { backendStatEntry 3 }

backendWeight OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend weight"
    ::= { backendStatEntry 4 }
```

```
backendOutstanding OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend outstanding queries"
    ::= { backendStatEntry 5 }

backendQPSLimit OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend QPS limit"
    ::= { backendStatEntry 6 }

backendReused OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend reused slots"
    ::= { backendStatEntry 7 }

backendState OBJECT-TYPE
    SYNTAX DisplayString
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend state"
    ::= { backendStatEntry 8 }

backendAddress OBJECT-TYPE
    SYNTAX OCTET STRING (SIZE (2..24))
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend address"
    ::= { backendStatEntry 9 }

backendPools OBJECT-TYPE
    SYNTAX DisplayString
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "List of pools this backend belongs to"
    ::= { backendStatEntry 10 }

backendQPS OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend QPS"
    ::= { backendStatEntry 11 }

backendQueries OBJECT-TYPE
    SYNTAX Counter64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Number of queries sent to this backend"
    ::= { backendStatEntry 12 }
```

```
backendOrder OBJECT-TYPE
    SYNTAX CounterBasedGauge64
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Backend order"
    ::= { backendStatEntry 13 }

---
--- Textual Conventions
---

SocketProtocolType ::= TEXTUAL-CONVENTION
    STATUS      current
    DESCRIPTION
        "A value that represents a type of socket protocol."
    SYNTAX       INTEGER {
                    unknown(0),
                    udp(1),
                    tcp(2)
                }

DNSQueryType ::= TEXTUAL-CONVENTION
    STATUS      current
    DESCRIPTION
        "A value that represents a type of DNS query (question or response)."
    SYNTAX       INTEGER {
                    unknown(0),
                    question(1),
                    response(2)
                }

---
--- Traps / Notifications
---

trap OBJECT IDENTIFIER ::= { dnsdist 10 }
traps OBJECT IDENTIFIER ::= { trap 0 } --- reverse-mappable
trapObjects OBJECT IDENTIFIER ::= { dnsdist 11 }

socketFamily OBJECT-TYPE
    SYNTAX InetAddressType
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Socket family type"
    ::= { trapObjects 1 }

socketProtocol OBJECT-TYPE
    SYNTAX SocketProtocolType
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Socket protocol type"
    ::= { trapObjects 2 }

fromAddress OBJECT-TYPE
    SYNTAX OCTET STRING (SIZE (2..24))
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Requestor address"
```

```
    ::= { trapObjects 3 }

toAddress OBJECT-TYPE
    SYNTAX OCTET STRING (SIZE (2..24))
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Responder address"
    ::= { trapObjects 4 }

queryType OBJECT-TYPE
    SYNTAX DNSQueryType
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Query / Response"
    ::= { trapObjects 5 }

querySize OBJECT-TYPE
    SYNTAX Unsigned32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "Size in bytes"
    ::= { trapObjects 6 }

queryID OBJECT-TYPE
    SYNTAX Unsigned32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "DNS query ID"
    ::= { trapObjects 7 }

qName OBJECT-TYPE
    SYNTAX OCTET STRING (SIZE (0..255))
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "DNS qname"
    ::= { trapObjects 8 }

qClass OBJECT-TYPE
    SYNTAX Unsigned32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "DNS query class"
    ::= { trapObjects 9 }

qType OBJECT-TYPE
    SYNTAX Unsigned32
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
        "DNS query type"
    ::= { trapObjects 10 }

trapReason OBJECT-TYPE
    SYNTAX OCTET STRING
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
```

```
        "Reason for this trap"
    ::= { trapObjects 11 }

backendStatusChangeTrap NOTIFICATION-TYPE
    OBJECTS {
        backendName,
        backendAddress,
        backendState
    }
    STATUS current
    DESCRIPTION "Backend status changed"
    ::= { traps 1 }

actionTrap NOTIFICATION-TYPE
    OBJECTS {
        socketFamily,
        socketProtocol,
        fromAddress,
        toAddress,
        queryType,
        querySize,
        queryID,
        qName,
        qClass,
        qType,
        trapReason
    }
    STATUS current
    DESCRIPTION "Trap sent by SNMPTrapAction"
    ::= { traps 2 }

customTrap NOTIFICATION-TYPE
    OBJECTS {
        trapReason
    }
    STATUS current
    DESCRIPTION "Trap sent by sendCustomTrap"
    ::= { traps 3 }

---
--- Conformance
---

dnsdistConformance OBJECT IDENTIFIER ::= { dnsdist 100 }

dnsdistCompliances MODULE-COMPLIANCE
    STATUS current
    DESCRIPTION "dnsdist compliance statement"
    MODULE
    MANDATORY-GROUPS {
        dnsdistGroup,
        dnsdistTrapsGroup
    }
    ::= { dnsdistConformance 1 }

dnsdistGroup OBJECT-GROUP
    OBJECTS {
        queries,
        responses,
        servfailResponses,
        aclDrops,
        ruleDrop,
        ruleNXDomain,
```

```
        ruleRefused,
        selfAnswered,
        downstreamTimeouts,
        downstreamSendErrors,
        truncFailures,
        noPolicy,
        latency01,
        latency110,
        latency1050,
        latency50100,
        latency1001000,
        latencySlow,
        latencyAVG100,
        latencyAVG1000,
        latencyAVG10000,
        latencyAVG1000000,
        uptime,
        realMemoryUsage,
        nonCompliantQueries,
        nonCompliantResponses,
        rdQueries,
        emptyQueries,
        cacheHits,
        cacheMisses,
        cpuUserMSec,
        cpuSysMSec,
        fdUsage,
        dynBlocked,
        dynBlockNMGSize,
        backendName,
        backendLatency,
        backendWeight,
        backendOutstanding,
        backendQPSLimit,
        backendReused,
        backendState,
        backendAddress,
        backendPools,
        backendQPS,
        backendQueries,
        backendOrder,
        socketFamily,
        socketProtocol,
        fromAddress,
        toAddress,
        queryType,
        querySize,
        queryID,
        qName,
        qClass,
        qType,
        trapReason
    }
    STATUS current
    DESCRIPTION "Objects conformance group for dnsdist"
    ::= { dnsdistConformance 2 }

dnsdistTrapsGroup NOTIFICATION-GROUP
    NOTIFICATIONS {
        actionTrap,
        customTrap,
        backendStatusChangeTrap
    }
```

```
    STATUS current
    DESCRIPTION "Traps conformance group for dnsdist"
    ::= { dnsdistConformance 3 }

END
```

14.10 AXFR, IXFR and NOTIFY

When **dnsdist** is deployed in front of a master authoritative server, it might receive AXFR or IXFR queries destined to this master. There are two issues that can arise in this kind of setup:

- If the master is part of a pool of servers, the first *SOA* query can be directed by **dnsdist** to a different server than the following AXFR/IXFR one, which might fail if the servers are not perfectly synchronised.
- If the master only allows AXFR/IXFR based on the source address of the requestor, it might be confused by the fact that the source address will be the one from the **dnsdist** server.

The first issue can be solved by routing SOA, AXFR and IXFR requests explicitly to the master:

```
newServer({address="192.168.1.2", name="master", pool={"master", "otherpool"}})
addAction(OrRule({QTypeRule(dnsdist.SOA), QTypeRule(dnsdist.AXFR),␣
↪QTypeRule(dnsdist.IXFR)}), PoolAction("master"))
```

The second one might require allowing AXFR/IXFR from the **dnsdist** source address and moving the source address check to **dnsdist**'s side:

```
addAction(AndRule({OrRule({QTypeRule(dnsdist.AXFR), QTypeRule(dnsdist.IXFR)}),␣
↪NotRule(makeRule("192.168.1.0/24"))}), RCodeAction(dnsdist.REFUSED))
```

When **dnsdist** is deployed in front of slaves, however, an issue might arise with NOTIFY queries, because the slave will receive a notification coming from the **dnsdist** address, and not the master's one. One way to fix this issue is to allow NOTIFY from the **dnsdist** address on the slave side (for example with PowerDNS's *trusted-notification-proxy*) and move the address check to **dnsdist**'s side:

```
addAction(AndRule({OpcodeRule(DNSOpcode.Notify), NotRule(makeRule("192.168.1.0/24
↪"))}), RCodeAction(dnsdist.REFUSED))
```

14.11 Running multiple instances

Sometimes, it can be advantageous to run multiple instances of **dnsdist**. Usecases can be:

- Multiple inbound IP addresses with different rulesets
- Taking advantage of more processes, using SO_REUSEPORT

dnsdist supports loading a different configuration file with the `--config` command line switch.

By default, `SYSCONFDIR/dnsdist.conf` is loaded. `SYSCONFDIR` is usually `/etc` or `/etc/dnsdist`.

14.11.1 Using systemd

New in version 1.3.0.

On systems with systemd, instance services can be used. To create a dnsdist service named `foo`, create a `dnsdist-foo.conf` in `SYSCONFDIR`, then run `systemctl enable dnsdist@foo.service` and `systemctl start dnsdist@foo.service`.

CHAPTER

FIFTEEN

REFERENCE GUIDES

These chapters contain extensive information on all functions and object available in dnsdist.

15.1 Configuration Reference

This page lists all configuration options for dnsdist.

Note: When an IPv6 IP:PORT combination is needed, the bracketed syntax from **RFC 3986** should be used. e.g. "[2001:DB8:14::C0FF:FEE]:5300".

15.1.1 Functions and Types

Within dnsdist several core object types exist:

- *Server*: generated with *newServer()*, represents a downstream server
- *ComboAddress*: represents an IP address and port
- *DNSName*: represents a domain name
- *NetmaskGroup*: represents a group of netmasks
- QPSLimiter: implements a QPS-based filter
- SuffixMatchNode: represents a group of domain suffixes for rapid testing of membership
- *DNSHeader*: represents the header of a DNS packet
- *ClientState*: sometimes also called Bind or Frontend, represents the addresses and ports dnsdist is listening on

The existence of most of these objects can mostly be ignored, unless you plan to write your own hooks and policies, but it helps to understand an expressions like:

```
getServer(0).order=12          -- set order of server 0 to 12
getServer(0):addPool("abuse")  -- add this server to the abuse pool
```

The `.` means `order` is a data member, while the `:` means `addPool` is a member function.

15.1.2 Global configuration

includeDirectory(*path*)
Include configuration files from `path`.

Parameters **path** (*str*) – The directory to load the configuration from

Listen Sockets

addLocal (*address*[, *options*])
New in version 1.2.0.

Add to the list of listen addresses.

Parameters

- **address** (*str*) – The IP Address with an optional port to listen on. The default port is 53.
- **options** (*table*) – A table with key: value pairs with listen options.

Options:

- `doTCP=true`: bool - Also bind on TCP on `address`.
- `reusePort=false`: bool - Set the `SO_REUSEPORT` socket option.
- `tcpFastOpenSize=0`: int - Set the TCP Fast Open queue size, enabling TCP Fast Open when available and the value is larger than 0.
- `interface=""`: str - Set the network interface to use.
- `cpus={}`: table - Set the CPU affinity for this listener thread, asking the scheduler to run it on a single CPU id, or a set of CPU ids. This parameter is only available if the OS provides the pthread_setaffinity_np() function.

```
addLocal('0.0.0.0:5300', { doTCP=true, reusePort=true })
```

This will bind to both UDP and TCP on port 5300 with SO_REUSEPORT enabled.

addLocal (*address*[[[, *do_tcp*], *so_reuseport*], *tcp_fast_open_qsize*])
Deprecated since version 1.2.0.

Add to the list of addresses listened on.

Parameters

- **address** (*str*) – The IP Address with an optional port to listen on. The default port is 53.
- **do_tcp** (*bool*) – Also bind a TCP port on `address`, defaults to true.
- **so_reuseport** (*bool*) – Use `SO_REUSEPORT` if it is available, defaults to false
- **tcp_fast_open_qsize** (*int*) – The size of the TCP Fast Open queue. Set to a number higher than 0 to enable TCP Fast Open when available. Default is 0.

setLocal (*address*[, *options*])
New in version 1.2.0.

Remove the list of listen addresses and add a new one.

Parameters

- **address** (*str*) – The IP Address with an optional port to listen on. The default port is 53.
- **options** (*table*) – A table with key: value pairs with listen options.

The options that can be set are the same as *addLocal()*.

setLocal (*address*[[[, *do_tcp*], *so_reuseport*], *tcp_fast_open_qsize*])
Deprecated since version 1.2.0.

Remove the list of listen addresses and add a new one.

Parameters

- **address** (*str*) – The IP Address with an optional port to listen on. The default port is 53.
- **do_tcp** (*bool*) – Also bind a TCP port on `address`, defaults to true.
- **so_reuseport** (*bool*) – Use `SO_REUSEPORT` if it is available, defaults to false
- **tcp_fast_open_qsize** (*int*) – The size of the TCP Fast Open queue. Set to a number higher than 0 to enable TCP Fast Open when available. Default is 0.

Control Socket, Console and Webserver

controlSocket(*address*)
Bind to `addr` and listen for a connection for the console

Parameters **address** (*str*) – An IP address with optional port. By default, the port is 5199.

inClientStartup()
Returns true while the console client is parsing the configuration.

makeKey()
Generate and print an encryption key.

setConsoleConnectionsLogging(*enabled*)
New in version 1.2.0.

Whether to log the opening and closing of console connections.

Parameters **enabled** (*bool*) – Default to true.

setKey(*key*)
Use `key` as shared secret between the client and the server

Parameters **key** (*str*) – An encoded key, as generated by *makeKey()*

testCrypto()
Test the crypto code, will report errors when something is not ok.

Webserver

webServer(*listen_address*, *password*[, *apikey*[, *custom_headers*]])
Launch the *Built-in webserver* with statistics and the API.

Parameters

- **listen_address** (*str*) – The IP address and Port to listen on
- **password** (*str*) – The password required to access the webserver
- **apikey** (*str*) – The key required to access the API
- **custom_headers** (*{[str]=str, ...}*) – Allows setting custom headers and removing the defaults

setAPIWritable(*allow*[, *dir*])
Allow modifications via the API. Optionally saving these changes to disk. Modifications done via the API will not be written to the configuration by default and will not persist after a reload

Parameters

- **allow** (*bool*) – Set to true to allow modification through the API
- **dir** (*str*) – A valid directory where the configuration files will be written by the API.

Access Control Lists

addACL (*netmask*)
Add a netmask to the existing ACL

Parameters **netmask** (*str*) – A CIDR netmask, e.g. `"192.0.2.0/24"`. Without a subnetmask, only the specific address is allowed.

setACL (*netmasks*)
Remove the existing ACL and add the netmasks from the table.

Parameters **netmasks** (*{str}*) – A table of CIDR netmask, e.g. `{"192.0.2.0/24", "2001:DB8:14::/56"}`. Without a subnetmask, only the specific address is allowed.

EDNS Client Subnet

setECSSourcePrefixV4 (*prefix*)
When `useClientSubnet` in *newServer()* is set and dnsdist adds an EDNS Client Subnet Client option to the query, truncate the requestors IPv4 address to `prefix` bits

Parameters **prefix** (*int*) – The prefix length

setECSSourcePrefixV6 (*prefix*)
When `useClientSubnet` in *newServer()* is set and dnsdist adds an EDNS Client Subnet Client option to the query, truncate the requestor's IPv6 address to bits

Parameters **prefix** (*int*) – The prefix length

Ringbuffers

setRingBuffersSize (*num*)
Set the capacity of the ringbuffers used for live traffic inspection to `num`

Parameters **num** (*int*) – The maximum amount of queries to keep in the ringbuffer. Defaults to 10000

15.1.3 Servers

newServer (*server_string*)
newServer (*server_table*)
Add a new backend server. Call this function with either a string:

```
newServer(
  "IP:PORT" -- IP and PORT of the backend server
)
```

or a table:

```
newServer({
  address="IP:PORT",      -- IP and PORT of the backend server (mandatory)
  qps=NUM,                -- Limit the number of queries per second to NUM,␣
↪when using the `firstAvailable` policy
  order=NUM,              -- The order of this server, used by the␣
↪`leastOustanding` and `firstAvailable` policies
  weight=NUM,             -- The weight of this server, used by the `wrandom`␣
↪and `whashed` policies
  pool=STRING|{STRING},   -- The pools this server belongs to (unset or empty␣
↪string means default pool) as a string or table of strings
  retries=NUM,            -- The number of TCP connection attempts to the␣
↪backend, for a given query
  tcpConnectTimeout=NUM, -- The timeout (in seconds) of a TCP connection␣
↪attempt
```

```
  tcpSendTimeout=NUM,     -- The timeout (in seconds) of a TCP write attempt
  tcpRecvTimeout=NUM,     -- The timeout (in seconds) of a TCP read attempt
  tcpFastOpen=BOOL,       -- Whether to enable TCP Fast Open
  ipBindAddrNoPort=BOOL, -- Whether to enable IP_BIND_ADDRESS_NO_PORT if␣
↪available, default: true
  name=STRING,            -- The name associated to this backend, for display␣
↪purpose
  checkClass=NUM,         -- Use NUM as QCLASS in the health-check query,␣
↪default: DNSClass.IN
  checkName=STRING,       -- Use STRING as QNAME in the health-check query,␣
↪default: "a.root-servers.net."
  checkType=STRING,       -- Use STRING as QTYPE in the health-check query,␣
↪default: "A"
  setCD=BOOL,             -- Set the CD (Checking Disabled) flag in the health-
↪check query, default: false
  maxCheckFailures=NUM,   -- Allow NUM check failures before declaring the␣
↪backend down, default: false
  mustResolve=BOOL,       -- Set to true when the health check MUST return a␣
↪NOERROR RCODE and an answer
  useClientSubnet=BOOL,   -- Add the client's IP address in the EDNS Client␣
↪Subnet option when forwarding the query to this backend
  source=STRING           -- The source address or interface to use for queries␣
↪to this backend, by default this is left to the kernel's address selection
                          -- The following formats are supported:
                          --   "address", e.g. "192.0.2.2"
                          --   "interface name", e.g. "eth0"
                          --   "address@interface", e.g. "192.0.2.2@eth0"
})
```

Parameters

- **server_string** (*str*) – A simple IP:PORT string.
- **server_table** (*table*) – A table with at least a 'name' key

getServer(*index*) → Server
Get a *Server*

Parameters **index** (*int*) – The number of the server (as seen in *showServers()*).

Returns The *Server* object or nil

getServers()
Returns a table with all defined servers.

rmServer(*index*)
rmServer(*server*)
Remove a backend server.

Parameters

- **index** (*int*) – The number of the server (as seen in *showServers()*).
- **server** (Server) – A *Server* object as returned by e.g. *getServer()*.

Server Functions

A server object returned by *getServer()* can be manipulated with these functions.

class Server
This object represents a backend server. It has several methods.

classmethod Server:**addPool**(*pool*)
Add this server to a pool.

Parameters **pool** (*str*) – The pool to add the server to

classmethod Server:**getName**() → string
Get the name of this server.

Returns The name of the server, or an empty string if it does not have one

classmethod Server:**getNameWithAddr**() → string
Get the name plus IP address and port of the server

Returns A string containing the server name if any plus the server address and port

classmethod Server:**getOutstanding**() → int
Get the number of outstanding queries for this server.

Returns The number of outstanding queries

classmethod Server:**isUp**() → bool
Returns the up status of the server

Returns true when the server is up, false otherwise

classmethod Server:**rmPool**(*pool*)
Removes the server from the named pool

Parameters **pool** (*str*) – The pool to remove the server from

classmethod Server:**setAuto**([*status*])

Changed in version 1.3.0: `status` optional parameter added.

Set the server in the default auto state. This will enable health check queries that will set the server `up` and `down` appropriately.

param bool status Set the initial status of the server to `up` (true) or `down` (false) instead of using the last known status

classmethod Server:**setQPS**(*limit*)
Limit the queries per second for this server.

Parameters **limit** (*int*) – The maximum number of queries per second

classmethod Server:**setDown**()
Set the server in an `DOWN` state. The server will not receive queries and the health checks are disabled

classmethod Server:**setUp**()
Set the server in an `UP` state. This server will still receive queries and health checks are disabled

Attributes

Server.**name**
The name of the server

Server.**upStatus**
Whether or not this server is up or down

Server.**order**
The order of the server

Server.**weight**
The weight of the server

15.1.4 Pools

*Server*s can be part of any number of pools. Pools are automatically created when a server is added to a pool (with *newServer()*), or can be manually created with *addPool()*.

addPool(*name*) → ServerPool
Returns a *ServerPool*.

Parameters **name** (*string*) – The name of the pool to create

getPool(*name*) → ServerPool
Returns a *ServerPool* or nil.

Parameters **name** (*string*) – The name of the pool

rmPool(*name*)

Remove the pool named *name*.

Parameters **name** (*string*) – The name of the pool to remove

getPoolServers(*name*) → [Server]
Returns a list of *Server*s or nil.

Parameters **name** (*string*) – The name of the pool

class ServerPool
This represents the pool where zero or more servers are part of.

classmethod ServerPool:**getCache**() → PacketCache
Returns the *PacketCache* for this pool or nil.

classmethod ServerPool:**setCache**(*cache*)
Adds `cache` as the pool's cache.

Parameters **cache** (PacketCache) – The new cache to add to the pool

classmethod ServerPool:**unsetCache**()
Removes the cache from this pool.

PacketCache

A Pool can have a packet cache to answer queries directly in stead of going to the backend. See *Caching Responses* for a how to.

newPacketCache(*maxEntries*[, *maxTTL=86400*[, *minTTL=0*[, *temporaryFailureTTL=60*[, *staleTTL=60*[, *dontAge=false*[, *numberOfShards=1*[, *deferrableInsertLock=true*]]]]]]]) → PacketCache
Changed in version 1.2.0: `numberOfShard` and `deferrableInsertLock` parameters added.

Creates a new *PacketCache* with the settings specified.

Parameters

- **maxEntries** (*int*) – The maximum number of entries in this cache
- **maxTTL** (*int*) – Cap the TTL for records to his number
- **minTTL** (*int*) – Don't cache entries with a TTL lower than this
- **temporaryFailureTTL** (*int*) – On a SERVFAIL or REFUSED from the backend, cache for this amount of seconds
- **staleTTL** (*int*) – When the backend servers are not reachable, send responses if the cache entry is expired at most this amount of seconds
- **dontAge** (*bool*) – Don't reduce TTLs when serving from the cache. Use this when **dnsdist** fronts a cluster of authoritative servers
- **numberOfShards** (*int*) – Number of shards to divide the cache into, to reduce lock contention
- **deferrableInsertLock** (*bool*) – Whether the cache should give up insertion if the lock is held by another thread, or simply wait to get the lock

class PacketCache
Represents a cache that can be part of *ServerPool*.

classmethod PacketCache:**expunge**(*n*)
Remove entries from the cache, leaving at most n entries

Parameters **n** (*int*) – Number of entries to keep

classmethod PacketCache:**expungeByName**(*name*[, *qtype=dnsdist.ANY*[, *suffixMatch=false*]])
Changed in version 1.2.0: suffixMatch parameter added.

Remove entries matching name and type from the cache.

Parameters

- **name** (DNSName) – The name to expunge
- **qtype** (*int*) – The type to expunge
- **suffixMatch** (*bool*) – When set to true, remove al entries under name

classmethod PacketCache:**isFull**() → bool
Return true if the cache has reached the maximum number of entries.

classmethod PacketCache:**printStats**()
Print the cache stats (hits, misses, deferred lookups and deferred inserts).

classmethod PacketCache:**purgeExpired**(*n*)
Remove expired entries from the cache until there is at most n entries remaining in the cache.

Parameters **n** (*int*) – Number of entries to keep

classmethod PacketCache:**toString**() → string
Return the number of entries in the Packet Cache, and the maximum number of entries

15.1.5 Client State

Also called frontend or bind, the Client State object returned by *getBind()* and listed with *showBinds()* represents an address and port dnsdist is listening on.

getBind(*index*) → ClientState
Return a ClientState object.

Parameters **index** (*int*) – The object index

ClientState functions

class ClientState
This object represents an address and port dnsdist is listening on. When reuseport is in use, several ClientState objects can be present for the same address and port.

classmethod Server:**addPool**(*pool*)
Add this server to a pool.

Parameters **pool** (*str*) – The pool to add the server to

classmethod ClientState:**attachFilter**(*filter*)
Attach a BPF filter to this frontend.

Parameters **filter** (BPFFilter) – The filter to attach to this frontend

classmethod ClientState:**detachFilter**()
Remove the BPF filter associated to this frontend, if any.

classmethod ClientState:**toString**() → string
Return the address and port this frontend is listening on.

Returns The address and port this frontend is listening on

Attributes

`ClientState.`**`muted`**
If set to true, queries received on this frontend will be normally processed and sent to a backend if needed, but no response will be ever be sent to the client over UDP. TCP queries are processed normally and responses sent to the client.

15.1.6 Status, Statistics and More

`dumpStats``()`
Print all statistics dnsdist gathers

`grepq`(*selector*[, *num*])
`grepq`(*selectors*[, *num*])
Prints the last `num` queries matching `selector` or `selectors`.

The selector can be:

- a netmask (e.g. '192.0.2.0/24')
- a DNS name (e.g. 'dnsdist.org')
- a response time (e.g. '100ms')

Parameters

- **`selector`** (*`str`*) – Select queries based on this property.
- **`selectors`** (*`{str}`*) – A lua table of selectors. Only queries matching all selectors are shown
- **`num`** (*`int`*) – Show a maximum of `num` recent queries, default is 10.

`showACL``()`
Print a list of all allowed netmasks.

`showBinds``()`
Print a list of all the current addresses and ports dnsdist is listening on, also called `frontends`

`showResponseLatency``()`
Show a plot of the response time latency distribution

`showServers``()`
This function shows all backend servers currently configured and some statistics. These statics have the following fields:

- `#` - The number of the server, can be used as the argument for *`getServer()`*
- `Address` - The IP address and port of the server
- `State` - The current state of the server
- `Qps` - Current number of queries per second
- `Qlim` - Configured maximum number of queries per second
- `Ord` - The order number of the server
- `Wt` - The weight of the server
- `Queries` - Total amount of queries sent to this server
- `Drops` - Number of queries that were dropped by this server
- `Drate` - Number of queries dropped per second by this server

- `Lat` - The latency of this server in milliseconds
- `Pools` - The pools this server belongs to

showTCPStats()
Show some statistics regarding TCP

showVersion()
Print the version of dnsdist

topBandwidth([*num*])
Print the top `num` clients that consume the most bandwidth.

Parameters **num** (*int*) – Number to show, defaults to 10.

topClients([*num*])
Print the top `num` clients sending the most queries over length of ringbuffer

Parameters **num** (*int*) – Number to show, defaults to 10.

topQueries([*num*[, *labels*]])
Print the `num` most popular QNAMEs from queries. Optionally grouped by the rightmost `labels` DNS labels.

Parameters

- **num** (*int*) – Number to show, defaults to 10
- **label** (*int*) – Number of labels to cut down to

topResponses([*num*[, *rcode*[, *labels*]]])
Print the `num` most seen responses with an RCODE of `rcode`. Optionally grouped by the rightmost `labels` DNS labels.

Parameters

- **num** (*int*) – Number to show, defaults to 10
- **rcode** (*int*) – *Response code*, defaults to 0 (No Error)
- **label** (*int*) – Number of labels to cut down to

topSlow([*num*[, *limit*[, *labels*]]])
Print the `num` slowest queries that are slower than `limit` milliseconds. Optionally grouped by the rightmost `labels` DNS labels.

Parameters

- **num** (*int*) – Number to show, defaults to 10
- **limit** (*int*) – Show queries slower than this amount of milliseconds, defaults to 2000
- **label** (*int*) – Number of labels to cut down to

15.1.7 Dynamic Blocks

addDynBlocks(*addresses*, *message*[, *seconds=10*[, *action*]])
Changed in version 1.2.0: `action` parameter added.

Block a set of addresses with `message` for (optionally) a number of seconds. The default number of seconds to block for is 10.

Parameters

- **addresses** – set of Addresses as returned by an exceed function
- **message** (*string*) – The message to show next to the blocks
- **seconds** (*int*) – The number of seconds this block to expire

- **action** (*int*) – The action to take when the dynamic block matches, see *here*. (default to the one set with `setDynBlocksAction()`)

clearDynBlocks()
Remove all current dynamic blocks.

showDynBlocks()
List all dynamic blocks in effect.

setDynBlocksAction(*action*)
Set which action is performed when a query is blocked. Only DNSAction.Drop (the default), DNSAction.Refused and DNSAction.Truncate are supported.

Getting addresses that exceeded parameters

exceedServFails(*rate*, *seconds*)
Get set of addresses that exceed `rate` servfails/s over `seconds` seconds

Parameters

- **rate** (*int*) – Number of Servfails per second to exceed
- **seconds** (*int*) – Number of seconds the rate has been exceeded

exceedNXDOMAINs(*rate*, *seconds*)
get set of addresses that exceed `rate` NXDOMAIN/s over `seconds` seconds

Parameters

- **rate** (*int*) – Number of NXDOMAIN per second to exceed
- **seconds** (*int*) – Number of seconds the rate has been exceeded

exceedRespByterate(*rate*, *seconds*)
get set of addresses that exceeded `rate` bytes/s answers over `seconds` seconds

Parameters

- **rate** (*int*) – Number of bytes per second to exceed
- **seconds** (*int*) – Number of seconds the rate has been exceeded

exceedQRate(*rate*, *seconds*)
Get set of address that exceed `rate` queries/s over `seconds` seconds

Parameters

- **rate** (*int*) – Number of queries per second to exceed
- **seconds** (*int*) – Number of seconds the rate has been exceeded

exceedQTypeRate(*type*, *rate*, *seconds*)
Get set of address that exceed `rate` queries/s for queries of QType `type` over `seconds` seconds

Parameters

- **type** (*int*) – QType
- **rate** (*int*) – Number of QType queries per second to exceed
- **seconds** (*int*) – Number of seconds the rate has been exceeded

15.1.8 Other functions

maintenance()
If this function exists, it is called every second to so regular tasks. This can be used for e.g. *Dynamic Blocks*.

15.2 Constants

There are many constants in **dnsdist**.

15.2.1 OPCode

- DNSOpcode.Query
- DNSOpcode.IQuery
- DNSOpcode.Status
- DNSOpcode.Notify
- DNSOpcode.Update

15.2.2 QClass

- QClass.IN
- QClass.CHAOS
- QClass.NONE
- QClass.ANY

15.2.3 RCode

- dnsdist.NOERROR
- dnsdist.FORMERR
- dnsdist.SERVFAIL
- dnsdist.NXDOMAIN
- dnsdist.NOTIMP
- dnsdist.REFUSED
- dnsdist.YXDOMAIN
- dnsdist.YXRRSET
- dnsdist.NXRRSET
- dnsdist.NOTAUTH
- dnsdist.NOTZONE

15.2.4 DNS Section

- DNSSection.Question
- DNSSection.Answer
- DNSSection.Authority
- DNSSection.Additional

15.2.5 DNSAction

These constants represent an Action that can be returned from the functions invoked by *addLuaAction()* and *addLuaResponseAction()*.

- `DNSAction.Allow`: let the query pass, skipping other rules
- `DNSAction.Delay`: delay the response for the specified milliseconds (UDP-only), continue to the next rule
- `DNSAction.Drop`: drop the query
- `DNSAction.HeaderModify`: indicate that the query has been turned into a response
- `DNSAction.None`: continue to the next rule
- `DNSAction.Nxdomain`: return a response with a NXDomain rcode
- `DNSAction.Pool`: use the specified pool to forward this query
- `DNSAction.Refused`: return a response with a Refused rcode
- `DNSAction.Spoof`: spoof the response using the supplied IPv4 (A), IPv6 (AAAA) or string (CNAME) value

15.3 ComboAddress

class ComboAddress
A `ComboAddress` represents an IP address with possibly a port number. The object can be an IPv4 or an IPv6 address.

Functions and methods related to `ComboAddress`

newCA(*address*) → :class:'ComboAddress'
Returns a *ComboAddress* based on `address`

Parameters **address** (*string*) – The IP address, with optional port, to represent.

classmethod `ComboAddress`**:getPort**() → int
Returns the port number.

classmethod `ComboAddress`**:isIPv4**() → bool
Returns true if the address is an IPv4, false otherwise

classmethod `ComboAddress`**:isIPv6**() → bool
Returns true if the address is an IPv6, false otherwise

classmethod `ComboAddress`**:isMappedIPv4**() → bool
Returns true if the address is an IPv4 mapped into an IPv6, false otherwise

classmethod `ComboAddress`**:mapToIPv4**() → ComboAddress
Convert an IPv4 address mapped in a v6 one into an IPv4. Returns a new ComboAddress

classmethod `ComboAddress`**:tostring**() → string
classmethod `ComboAddress`**:toString**() → string
Returns in human-friendly format

classmethod `ComboAddress`**:tostringWithPort**() → string
classmethod `ComboAddress`**:toStringWithPort**() → string
Returns in human-friendly format, with port number

classmethod `ComboAddress`**:truncate**(*bits*)
Truncate the ComboAddress to the specified number of bits. This essentially zeroes all bits after `bits`.

Parameters **bits** (*int*) – Amount of bits to truncate to

15.4 NetmaskGroup

class NetmaskGroup
Represents a group of netmasks that can be used to match *ComboAddress*es against.

newNMG() → NetmaskGroup
Returns a NetmaskGroup

classmethod NetmaskGroup:**addMask**(*mask*)
classmethod NetmaskGroup:**addMask**(*masks*)
Add one or more masks to the NMG.

Parameters

- **mask** (*string*) – Add this mask, prefix with *!* to exclude this mask from matching.
- **masks** (*table*) – Adds the keys of the table to the *NetmaskGroup*. It should be a table whose keys are *ComboAddress* objects and values are integers, as returned by *exceed** functions.

classmethod NetmaskGroup:**match**(*address*) → bool
Checks if `address` is matched by this NetmaskGroup.

Parameters **address** (ComboAddress) – The address to match.

classmethod NetmaskGroup:**clear**()
Clears the NetmaskGroup.

classmethod NetmaskGroup:**size**() → int
Returns number of netmasks in this NetmaskGroup.

15.5 DNSName objects

class DNSName
A `DNSName` object represents a name in the DNS. It is returned by several functions and has several functions to programmatically interact with it.

Creating a `DNSName` is done with the *newDNSName()*:

```
myname = newDNSName("www.example.com")
```

dnsdist will complain loudly if the name is invalid (e.g. too long, dot in the wrong place).

The `myname` variable has several functions to get information from it

```
print(myname:countLabels()) -- prints "3"
print(myname:wirelength()) -- prints "17"
name2 = newDNSName("example.com")
if myname:isPartOf(name2) then -- prints "it is"
  print('it is')
end
```

15.5.1 Functions and methods of a DNSName

newDNSName(*name*) → DNSName
Returns the *DNSName* object of `name`.

Parameters **name** (*string*) – The name to create a DNSName for

DNSName::chopoff() -> bool
New in version 1.2.0.

Removes the left-most label and returns `true`. `false` is returned if no label was removed

classmethod DNSName:**countLabels**() → int
Returns the number of DNSLabels in the name

classmethod DNSName:**isPartOf**(*name*) → bool
Returns true if the DNSName is part of the DNS tree of `name`.

Parameters **name** (DNSName) – The name to check against

classmethod DNSName:**toString**() → string
classmethod DNSName:**tostring**() → string
Returns a human-readable form of the DNSName.

DNSName:wirelength -> int
Returns the length in bytes of the DNSName as it would be on the wire.

15.6 The DNSQuestion (dq) object

A DNSQuestion or `dq` object is available in several hooks and Lua actions. This object contains details about the current state of the question. This state can be modified from the various hooks.

The DNSQuestion object has several attributes, many of them read-only:

class DNSQuestion

DNSQuestion.**dh**
The *DNSHeader (dh) object* of this query.

DNSQuestion.**ecsOverride**
Whether an existing ECS value should be overridden, settable.

DNSQuestion.**ecsPrefixLength**
The ECS prefix length to use, settable.

DNSQuestion.**len**
The length of the *qname*.

DNSQuestion.**localaddr**
ComboAddress of the local bind this question was received on.

DNSQuestion.**opcode**
Integer describing the OPCODE of the packet. Can be matched against *OPCode*.

DNSQuestion.**qclass**
QClass (as an unsigned integer) of this question. Can be compared against *QClass*.

DNSQuestion.**qname**
DNSName of this question.

DNSQuestion.**qtype**
QType (as an unsigned integer) of this question. Can be compared against `dnsdist.A`, `dnsdist.AAAA` etc.

DNSQuestion.**remoteaddr**
ComboAddress of the remote client.

DNSQuestion.**rcode**
RCode (as an unsigned integer) of this question. Can be compared against *RCode*

DNSQuestion.**size**
The total size of the buffer starting at *DNSQuestion.dh*.

DNSQuestion.**skipCache**
Whether to skip cache lookup / storing the answer for this question, settable.

DNSQuestion.**tcp**
Whether the query have been received over TCP.

`DNSQuestion.`**`useECS`**
Whether to send ECS to the backend, settable.

It also supports the following methods:

classmethod `DNSQuestion:`**`getDO`**`()` → bool
New in version 1.2.0.

Get the value of the DNSSEC OK bit.

Returns true if the DO bit was set, false otherwise

classmethod `DNSQuestion:`**`getTag`**(*key*) → string
New in version 1.2.0.

Get the value of a tag stored into the DNSQuestion object.

Parameters **`key`** (*`string`*) – The tag's key

Returns A table of tags, using strings as keys and values

classmethod `DNSQuestion:`**`getTagArray`**`()` → table
New in version 1.2.0.

Get all the tags stored into the DNSQuestion object.

Returns The tag's value if it was set, an empty string otherwise

classmethod `DNSQuestion:`**`sendTrap`**(*reason*)
New in version 1.2.0.

Send an SNMP trap.

Parameters **`reason`** (*`string`*) – An optional string describing the reason why this trap was sent

classmethod `DNSQuestion:`**`setTag`**(*key*, *value*)
New in version 1.2.0.

Set a tag into the DNSQuestion object.

Parameters

- **`key`** (*`string`*) – The tag's key
- **`value`** (*`string`*) – The tag's value

classmethod `DNSQuestion:`**`setTagArray`**(*tags*)
New in version 1.2.0.

Set an array of tags into the DNSQuestion object.

Parameters **`tags`** (*`table`*) – A table of tags, using strings as keys and values

15.7 DNSResponse object

class DNSResponse
This object has all the functions and members of a *DNSQuestion* and some more

classmethod `DNSResponse:`**`editTTLs`**(*func*)
The function `func` is invoked for every entry in the answer, authority and additional section.

`func` points to a function with the following prototype: `myFunc(section, qclass, qtype, ttl)`

All parameters to `func` are integers:

- `section` is the section in the packet and can be compared to *DNS Section*
- `qclass` is the QClass of the record. Can be compared to *QClass*

- `qtype` is the QType of the record. Can be e.g. compared to `dnsdist.A`, `dnsdist.AAAA` and the like.
- `ttl` is the current TTL

This function must return an integer with the new TTL. Setting this TTL to 0 to leaves it unchanged

Parameters **`func`** (*`string`*) – The function to call to edit TTLs.

15.8 DNSHeader (`dh`) object

`class DNSHeader`
This object holds a representation of a DNS packet's header.

`classmethod` `DNSHeader:`**`getRD`**`()` → bool
Get recursion desired flag.

`classmethod` `DNSHeader:`**`setRD`**(*rd*)
Set recursion desired flag.

Parameters **`rd`** (*`bool`*) – State of the RD flag

`classmethod` `DNSHeader:`**`setTC`**(*tc*)
Set truncation flag (TC).

Parameters **`tc`** (*`bool`*) – State of the TC flag

`classmethod` `DNSHeader:`**`setQR`**(*qr*)
Set Query/Response flag. Setting QR to true means "This is an answer packet".

Parameters **`qr`** (*`bool`*) – State of the QR flag

`classmethod` `DNSHeader:`**`getCD`**`()` → bool
Get checking disabled flag.

`classmethod` `DNSHeader:`**`setCD`**(*cd*)
Set checking disabled flag.

Parameters **`cd`** (*`bool`*) – State of the CD flag

15.9 eBPF functions and objects

These are all the functions, objects and methods related to the *eBPF Socket Filtering*.

`addBPFFilterDynBlocks`(*addresses*, *dynbpf*[, *seconds=10*])
This is the eBPF equivalent of *`addDynBlocks()`*, blocking a set of addresses for (optionally) a number of seconds, using an eBPF dynamic filter. The default number of seconds to block for is 10.

Parameters

- **`addresses`** – set of Addresses as returned by an *exceed function*
- **`dynbpf`** (`DynBPFFilter`) – The dynamic eBPF filter to use
- **`seconds`** (*`int`*) – The number of seconds this block to expire

`newBPFFilter`(*maxV4*, *maxV6*, *maxQNames*) → BPFFilter
Return a new eBPF socket filter with a maximum of maxV4 IPv4, maxV6 IPv6 and maxQNames qname entries in the block table.

Parameters

- **`maxV4`** (*`int`*) – Maximum number of IPv4 entries in this filter
- **`maxV6`** (*`int`*) – Maximum number of IPv6 entries in this filter

- **maxQNames** (*int*) – Maximum number of QName entries in this filter

newDynBPFFilter(*bpf*) → DynBPFFilter
Return a new dynamic eBPF filter associated to a given BPF Filter.

Parameters **bpf** (BPFFilter) – The underlying eBPF filter

setDefaultBPFFilter(*filter*)
When used at configuration time, the corresponding BPFFilter will be attached to every bind.

Parameters **filter** (BPFFilter) – The filter to attach

registerDynBPFFilter(*dynbpf*)

Register a DynBPFFilter filter so that it appears in the web interface and the API.

Parameters **dynbpf** (DynBPFFilter) – The dynamic eBPF filter to register

unregisterDynBPFFilter(*dynbpf*)

Remove a DynBPFFilter filter from the web interface and the API.

Parameters **dynbpf** (DynBPFFilter) – The dynamic eBPF filter to unregister

class BPFFilter
Represents an eBPF filter

classmethod BPFFilter:**attachToAllBinds**()
Attach this filter to every bind already defined. This is the run-time equivalent of *setDefaultBPFFilter()*

classmethod BPFFilter:**block**(*address*)
Block this address

Parameters **address** (ComboAddress) – The address to block

classmethod BPFFilter:**blockQName**(*name*[, *qtype=255*])
Block queries for this exact qname. An optional qtype can be used, defaults to 255.

Parameters

- **name** (DNSName) – The name to block
- **qtype** (*int*) – QType to block

classmethod BPFFilter:**getStats**()
Print the block tables.

classmethod BPFFilter:**unblock**(*address*)
Unblock this address.

Parameters **address** (ComboAddress) – The address to unblock

classmethod BPFFilter:**unblockQName**(*name*[, *qtype=255*])
Remove this qname from the block list.

Parameters

- **name** (DNSName) – the name to unblock
- **qtype** (*int*) – The qtype to unblock

class DynBPFFilter
Represents an dynamic eBPF filter, allowing the use of ephemeral rules to an existing eBPF filter.

classmethod BPFFilter:**purgeExpired**()
Remove the expired ephemeral rules associated with this filter.

15.10 DNSCrypt objects and functions

addDNSCryptBind(*address*, *provider*, *certificate*, *keyfile*[, *options*])
Adds a DNSCrypt listen socket on `address`.

Parameters

- **address** (*string*) – The address and port to listen on
- **provider** (*string*) – The provider name for this bind
- **certificate** (*string*) – Path to the certificate file
- **keyfile** (*string*) – Path to the key file of the certificate
- **options** (*table*) – A table with key: value pairs with options (see below)

Options:

- `doTCP=true`: bool - Also bind on TCP on `address`.
- `reusePort=false`: bool - Set the `SO_REUSEPORT` socket option.
- `tcpFastOpenSize=0`: int - Set the TCP Fast Open queue size, enabling TCP Fast Open when available and the value is larger than 0
- `interface=""`: str - Sets the network interface to use
- `cpus={}`: table - Set the CPU affinity for this listener thread, asking the scheduler to run it on a single CPU id, or a set of CPU ids. This parameter is only available if the OS provides the pthread_setaffinity_np() function.

generateDNSCryptProviderKeys(*publicKey*, *privateKey*)
Generate a new provider keypair and write them to `publicKey` and `privateKey`.

Parameters

- **publicKey** (*string*) – path to write the public key to
- **privateKey** (*string*) – path to write the private key to

generateDNSCryptCertificate(*privatekey*, *certificate*, *keyfile*, *serial*, *validFrom*, *validUntil*)
generate a new resolver private key and related certificate, valid from the `validFrom` UNIX timestamp until the `validUntil` one, signed with the provider private key.

Parameters

- **privatekey** (*string*) – Path to the private key of the provider.
- **certificate** (*string*) – Path where to write the certificate file.
- **keyfile** (*string*) – Path where to write the private key for the certificate.
- **serial** (*int*) – The certificate's serial number.
- **validFrom** (*int*) – Unix timestamp from when the certificate will be valid.
- **validUntil** (*int*) – Unix timestamp until when the certificate will be valid.

printDNSCryptProviderFingerprint(*keyfile*)
Display the fingerprint of the provided resolver public key

Parameters **keyfile** (*string*) – Path to the key file

showDNSCryptBinds()
Display the currently configured DNSCrypt binds

getDNSCryptBind(*n*) → DNSCryptContext
Return the *DNSCryptContext* object corresponding to the bind `n`.

15.10.1 Certificates

class DNSCryptCert
Represents a DNSCrypt certificate.

classmethod DNSCryptCert:**getClientMagic**() → string
Return this certificate's client magic value.

classmethod DNSCryptCert:**getEsVersion**() → string
Return the cryptographic construction to use with this certificate,.

classmethod DNSCryptCert:**getMagic**() → string
Return the certificate magic number.

classmethod DNSCryptCert:**getProtocolMinorVersion**() → string
Return this certificate's minor version.

classmethod DNSCryptCert:**getResolverPublicKey**() → string
Return the public key corresponding to this certificate.

classmethod DNSCryptCert:**getSerial**() → int
Return the certificate serial number.

classmethod DNSCryptCert:**getSignature**() → string
Return this certificate's signature.

classmethod DNSCryptCert:**getTSEnd**() → int
Return the date the certificate is valid from, as a Unix timestamp.

classmethod DNSCryptCert:**getTSStart**() → int
Return the date the certificate is valid until (inclusive), as a Unix timestamp

15.10.2 Context

class DNSCryptContext
Represents a DNSCrypt content. Can be used to rotate certs.

classmethod DNSCryptContext:**generateAndLoadInMemoryCertificate**(*keyfile*, *serial*, *begin*, *end*)
Generate a new resolver key and the associated certificate in-memory, sign it with the provided provider key, and use the new certificate

Parameters

- **keyfile** (*string*) – Path to the key file to use
- **serial** (*int*) – The serial number of the certificate
- **begin** (*int*) – Unix timestamp from when the certificate is valid
- **end** (*int*) – Unix timestamp from until the certificate is valid

classmethod DNSCryptContext:**getCurrentCertificate**() → DNSCryptCert
Return the current certificate.

classmethod DNSCryptContext:**getOldCertificate**() → DNSCryptCert
Return the previous certificate.

classmethod DNSCryptContext:**getProviderName**() → string
Return the provider name

classmethod DNSCryptContext:**hasOldCertificate**() → bool
Whether or not the context has a previous certificate, from a certificate rotation.

classmethod DNSCryptContext:**loadNewCertificate**(*certificate*, *keyfile*)
Load a new certificate and the corresponding private key, and use it

Parameters

- **certificate** (*string*) – Path to a certificate file
- **keyfile** (*string*) – Path to a the corresponding key file

15.11 Protobuf Logging Reference

newRemoteLogger(*address*[, *timeout=2*[, *maxQueuedEntries=100*[, *reconnectWaitTime=1*]]])
Create a Remote Logger object, to use with *RemoteLogAction()* and *RemoteLogResponseAction()*.

Parameters

- **address** (*string*) – An IP:PORT combination where the logger is listening
- **timeout** (*int*) – TCP connect timeout in seconds
- **maxQueuedEntries** (*int*) – Queue this many messages before dropping new ones (e.g. when the remote listener closes the connection)
- **reconnectWaitTime** (*int*) – Time in seconds between reconnection attempts

class DNSDistProtoBufMessage
This object represents a single protobuf message as emitted by **dnsdist**.

classmethod DNSDistProtoBufMessage:**addResponseRR**(*name*, *type*, *class*, *ttl*, *blob*)
New in version 1.2.0.

Add a response RR to the protobuf message.

Parameters

- **name** (*string*) – The RR name.
- **type** (*int*) – The RR type.
- **class** (*int*) – The RR class.
- **ttl** (*int*) – The RR TTL.
- **blob** (*string*) – The RR binary content.

classmethod DNSDistProtoBufMessage:**setBytes**(*bytes*)
Set the size of the query

Parameters **bytes** (*int*) – Number of bytes in the query.

classmethod DNSDistProtoBufMessage:**setEDNSSubnet**(*netmask*)
Set the EDNS Subnet to `netmask`.

Parameters **netmask** (*string*) – The netmask to set to.

classmethod DNSDistProtoBufMessage:**setQueryTime**(*sec*, *usec*)
In a response message, set the time at which the query has been received.

Parameters

- **sec** (*int*) – Unix timestamp when the query was received.
- **usec** (*int*) – The microsecond the query was received.

classmethod DNSDistProtoBufMessage:**setQuestion**(*name*, *qtype*, *qclass*)
Set the question in the protobuf message.

Parameters

- **name** (DNSName) – The qname of the question
- **qtype** (*int*) – The qtype of the question

- **qclass** (*int*) – The qclass of the question

classmethod DNSDistProtoBufMessage:**setProtobufResponseType**(*sec*, *usec*)
New in version 1.2.0.

Change the protobuf response type from a query to a response, and optionally set the query time.

Parameters

- **sec** (*int*) – Optional query time in seconds.
- **usec** (*int*) – Optional query time in additional micro-seconds.

classmethod DNSDistProtoBufMessage:**setRequestor**(*address*)
Set the requestor's address.

Parameters **address** (ComboAddress) – The address to set to

classmethod DNSDistProtoBufMessage:**setRequestorFromString**(*address*)
Set the requestor's address from a string.

Parameters **address** (*string*) – The address to set to

classmethod DNSDistProtoBufMessage:**setResponder**(*address*)
Set the responder's address.

Parameters **address** (ComboAddress) – The address to set to

classmethod DNSDistProtoBufMessage:**setResponderFromString**(*string*)
Set the responder's address.

Parameters **address** (*string*) – The address to set to

classmethod DNSDistProtoBufMessage:**setResponseCode**(*rcode*)
Set the response code of the query.

Parameters **rcode** (*int*) – The response code of the answer

classmethod DNSDistProtoBufMessage:**setTag**(*value*)
New in version 1.2.0.

Add a tag to the list of tags.

Parameters **value** (*string*) – The tag value

classmethod DNSDistProtoBufMessage:**setTagArray**(*valueList*)
New in version 1.2.0.

Add a list of tags.

Parameters **tags** (*table*) – A list of tags as strings

classmethod DNSDistProtoBufMessage:**setTime**(*sec*, *usec*)
Set the time at which the query or response has been received.

Parameters

- **sec** (*int*) – Unix timestamp when the query was received.
- **usec** (*int*) – The microsecond the query was received.

classmethod DNSDistProtoBufMessage:**toDebugString**() → string
Return an string containing the content of the message

15.12 Carbon export

carbonServer(*serverIP*[, *ourname*][, *interval*])
Exort statistics to a Carbon / Graphite / Metronome server.

Parameters

- **serverIP** (*string*) – Indicates the IP address where the statistics should be sent
- **ourname** (*string*) – An optional string specifying the hostname that should be used
- **interval** (*int*) – An optional unsigned integer indicating the interval in seconds between exports

15.13 SNMP reporting

New in version 1.2.0.

snmpAgent (*enableTraps*[, *masterSocket*])
Enable SNMP support.

Parameters

- **enableTraps** (*bool*) – Indicates whether traps should be sent
- **masterSocket** (*string*) – A string specifying how to connect to the master agent. This is a file path to a unix socket, but e.g. `tcp:localhost:705` can be used as well. By default, SNMP agent's default socket is used.

sendCustomTrap (*message*)
Send a custom SNMP trap from Lua.

Parameters **message** (*string*) – The message to include in the sent trap

15.14 Tuning related functions

setMaxTCPClientThreads (*num*)
Set the maximum of TCP client threads, handling TCP connections

Parameters **num** (*int*) –

setMaxTCPConnectionDuration (*num*)
Set the maximum duration of an incoming TCP connection, in seconds. 0 (the default) means unlimited

Parameters **num** (*int*) –

setMaxTCPConnectionsPerClient (*num*)
Set the maximum number of TCP connections per client. 0 (the default) means unlimited

Parameters **num** (*int*) –

setMaxTCPQueriesPerConnection (*num*)
Set the maximum number of queries in an incoming TCP connection. 0 (the default) means unlimited

Parameters **num** (*int*) –

setMaxTCPQueuedConnections (*num*)
Set the maximum number of TCP connections queued (waiting to be picked up by a client thread), defaults to 1000. 0 means unlimited

Parameters **num** (*int*) –

setMaxUDPOutstanding (*num*)
Set the maximum number of outstanding UDP queries to a given backend server. This can only be set at configuration time and defaults to 10240

Parameters **num** (*int*) –

setCacheCleaningDelay (*num*)
Set the interval in seconds between two runs of the cache cleaning algorithm, removing expired entries

Parameters **num** (*int*) –

setCacheCleaningPercentage(*num*)
Set the percentage of the cache that the cache cleaning algorithm will try to free by removing expired entries. By default (100), all expired entries are removed

Parameters **num** (*int*) –

setStaleCacheEntriesTTL(*num*)
Allows using cache entries expired for at most n seconds when no backend available to answer for a query

Parameters **num** (*int*) –

setTCPUseSinglePipe(*val*)
Whether the incoming TCP connections should be put into a single queue instead of using per-thread queues. Defaults to false

Parameters **val** (*bool*) –

setTCPRecvTimeout(*num*)
Set the read timeout on TCP connections from the client, in seconds

Parameters **num** (*int*) –

setTCPSendTimeout(*num*)
Set the write timeout on TCP connections from the client, in seconds

Parameters **num** (*int*) –

setUDPMultipleMessagesVectorSize(*num*)
New in version 1.2.0.

Set the maximum number of UDP queries messages to accept in a single *recvmmsg()* call. Only available if the underlying OS support *recvmmsg()* with the *MSG_WAITFORONE* option. Defaults to 1, which means only query at a time is accepted, using *recvmsg()* instead of *recvmmsg()*.

Parameters **num** (*int*) –

setUDPTimeout(*num*)
Set the maximum time dnsdist will wait for a response from a backend over UDP, in seconds. Defaults to 2

Parameters **num** (*int*) –

CHAPTER

SIXTEEN

MANUAL PAGES

16.1 dnsdist

16.1.1 Synopsis

dnsdist [<option>...] [address]...

16.1.2 Description

dnsdist receives DNS queries and relays them to one or more downstream servers. It subsequently sends back responses to the original requestor.

dnsdist operates over TCP and UDP, and strives to deliver very high performance over both.

Currently, queries are sent to the downstream server with the least outstanding queries. This effectively implies load balancing, making sure that slower servers get less queries.

If a reply has not come in after a few seconds, it is removed from the queue, but in the short term, timeouts do cause a server to get less traffic.

IPv4 and IPv6 operation can be mixed and matched, in other words, queries coming in over IPv6 could be forwarded to IPv4 and vice versa.

dnsdist is scriptable in Lua, see the dnsdist documentation for more information on this.

16.1.3 Scope

dnsdist does not 'think' about DNS queries, it restricts itself to measuring response times and error codes and routing questions accordingly. It comes with a very high performance packet-cache.

The goal for dnsdist is to remain simple. If more powerful loadbalancing is required, dedicated hardware or software is recommended. Linux Virtual Server for example is often mentioned.

16.1.4 Options

-a <netmask>, --acl <netmask> Add *netmask* to the ACL.

-C <file>, --config <file> Load configuration from *file*.

--check-config Test the configuration file (which may be set with **–config** or **-C**) for errors. dnsdist will show the errors and exit with a non-zero exit-code when errors are found.

-c <address>, --client <address> Operate as a client, connect to dnsdist. This will read the dnsdist configuration for the **controlSocket** statement and connect to it. When *address* (with an optional port number) is set, dnsdist will connect to that instead.

-k <key>, --setkey <key> When operating as a client(**-c**, **–client**), use *key* as shared secret to connect to dnsdist. This should be the same key that is used on the server (set with **setKey()**). Note that this will leak the key into your shell's history. Only available when dnsdist is compiled with libsodium support.

-d, --daemon Operate as a daemon.

-e, --execute <command> Connect to dnsdist and execute *command*.

-h, --help Display a helpful message and exit.

-l, --local <address> Bind to *address*, Supply as many addresses (using multiple **–local** statements) to listen on as required. Specify IPv4 as 0.0.0.0:53 and IPv6 as [::]:53.

--supervised Run in foreground, but do not spawn a console. Use this switch to run dnsdist inside a supervisor (use with e.g. systemd and daemontools).

--disable-syslog Disable logging to syslog. Use this when running inside a supervisor that handles logging (like systemd). Do not use in combination with **–daemon**.

-p, --pidfile <file> Write a pidfile to *file*, works only with **–daemon**.

-u, --uid <uid> Change the process user to *uid* after binding sockets. *uid* can be a name or number.

-g, --gid <gid> Change the process group to *gid* after binding sockets. *gid* Can be a name or number.

-V, --version Show the dnsdist version and exit.

-v, --verbose Be verbose.

address are any number of downstream DNS servers, in the same syntax as used with **–local**. If the port is not specified, 53 is used.

16.1.5 Bugs

Right now, the TCP support has some rather arbitrary limits.

16.1.6 Resources

Website: http://dnsdist.org

CHAPTER

SEVENTEEN

CHANGELOG

17.1 1.2.0

Released: 21st of August 2017

17.1.1 New Features

- Add an option to export CNAME records over protobuf. ¶ References: #4709, pull request 4776
- Add TCP management options from **RFC 7766 section 10**. ¶ References: pull request 4611
- Add an option to 'mute' UDP responses per bind. ¶ References: #4527, pull request 4536
- Save history to home-dir, only use CWD as a last resort. ¶ References: #4562, pull request 4779
- Add the `setRingBuffersSize()` directive to allows changing the ringbuffer size. ¶ References: pull request 4898
- Allow TTL alteration via Lua. ¶ References: #4707, pull request 4787
- Add `RDRule()` to match queries with the `RD` flag set. ¶ References: pull request 4837
- Add `setWHashedPertubation()` for consistent `whashed` results. ¶ References: pull request 4897
- Add `tcpConnectTimeout` to `newServer()`. ¶ References: pull request 4818
- Add cache hit response rules. ¶ References: #4708, pull request 4788, pull request 5036
- Add *SNMP support*. ¶ References: pull request 4989, pull request 5123, pull request 5204
- Allow passing `DNSName`s as DNSRules. ¶ References: pull request 5070
- Add support for setting the server selection policy on a per pool basis (Robin Geuze). ¶ References: pull request 5113
- Add a `suffixMatch` parameter to `PacketCache:expungeByName()` (Robin Geuze). ¶ References: pull request 5159
- Add an option so the packet cache entries don't age. ¶ References: #5126, pull request 5136
- Add `QNameRule()`. ¶ References: pull request 5235
- Add an optional action to `addDynBlocks()`. ¶ References: pull request 5337
- Add an optional interface parameter to `addLocal()`/`setLocal()`. ¶ References: pull request 5344
- Make a `truncate` action available to DynBlock and Lua. ¶ References: pull request 5386
- Implement a runtime changeable rule that matches IP address for a certain time called `TimedIPSetRule()`. ¶ References: pull request 5336
- Add support for returning several IPs to spoof from Lua. ¶ References: pull request 5496
- Add Lua bindings to be able to rotate DNSCrypt keys, see *DNSCrypt*. ¶ References: #5507, #5420, pull request 5490, pull request 5508

- Add the capability to set arbitrary tags in protobuf messages. ¶ References: pull request 5396, pull request 5577
- Add setConsoleConnectionsLogging(). ¶ References: #5565, pull request 5581

17.1.2 Improvements

- Merge the client and server nonces to prevent replay attacks. ¶ References: pull request 4815
- Store the computed shared key and reuse it for the response for DNSCrypt messages. ¶ References: pull request 4813, pull request 4926
- Add *setTCPUseSinglePipe()* to use a single TCP waiting queue. ¶ References: pull request 4817
- Add `sendSizeAndMsgWithTimeout` to send size and data in a single call and use it for TCP Fast Open towards backends. ¶ References: #5494, pull request 5501, pull request 4985
- Tune systemd unit-file for medium-sized installations (Winfried Angele). ¶ References: pull request 4958
- Add the possiblity to fill a *NetmaskGroup* (using *NetmaskGroup:addMask()*) from *exceeds** results. ¶ References: pull request 5185
- Add labels count to StatNode, only set the name once. ¶ References: pull request 5353
- DNSName: Check that both first two bits are set in compressed labels. ¶ References: #4851, pull request 4852
- Handle unreachable servers at startup, reconnect stale sockets ¶ References: #4155, #4131, pull request 4285
- Gracefully handle invalid addresses in *newServer()*. ¶ References: #4471, pull request 4474
- Use `IP_BIND_ADDRESS_NO_PORT` when available. ¶ References: pull request 4786
- Add an optional `seconds` parameter to `statNodeRespRing()`. ¶ References: #4775, #4660, pull request 4780
- Report a more specific lua version and report luajit in `--version`. ¶ References: pull request 4910
- Prevent issues by unshadowing variables. ¶ References: pull request 5056
- Register DNSName::chopOff (@plzz). ¶ References: pull request 4920
- Make *includeDirectory()* work sorted (Robin Geuze). ¶ References: #5053, pull request 5171, pull request 5150
- Allow embedded NULs in strings received from Lua. ¶ References: pull request 5147
- Cleanup closed TCP downstream connections. ¶ References: pull request 5163
- Improve reporting of C++ exceptions that bubble up via Lua. ¶ References: pull request 5230
- Add better logging on queries that get dropped, timed out or received. ¶ References: pull request 5253
- Print useful messages when query and response actions are mixed. ¶ References: pull request 5342
- Add `DNSRule::toString()` and add virtual destructors to DNSRule, DNSAction and DNSResponseAction so the destructors of derived classes are run even when deleted via the base type. ¶ References: pull request 5497
- Don't use square brackets for IPv6 in Carbon metrics. ¶ References: #5538, pull request 5579

17.1.3 Bug Fixes

- Unified `-k` and *setKey()* behaviour for client and server mode now. ¶ References: pull request 5199
- Refactor SuffixMatchNode using a SuffixMatchTree. ¶ References: #4761, pull request 4950
- Get rid of `std::move()` calls preventing copy elision. ¶ References: pull request 5359

- Send an HTTP 404 on unknown API paths. ¶ References: pull request 5089
- LuaWrapper: Use the correct index when storing a function. ¶ References: pull request 4775
- Send a latency of 0 over carbon, null over API for down servers. ¶ References: #4689, pull request 4785
- Fix negative port detection for IPv6 addresses on 32-bit. ¶ References: pull request 4911
- Fix crashed on SmartOS/Illumos (Roman Dayneko). ¶ References: #4579, pull request 4877
- Change `truncateTC` to defaulting to off, having it enabled by default causes an compatibility with **RFC 6891** (Robin Geuze). ¶ References: #4857, pull request 4859
- Don't cache answers without any TTL (like SERVFAIL). ¶ References: #4983, pull request 5037, pull request 4987
- Fix destination port reporting on "any" binds. ¶ References: pull request 5194
- Correctly truncate EDNS Client Subnetmasks. ¶ References: pull request 5320
- Fix *`RecordsTypeCountRule()`*'s handling of the # of records in a section. ¶ References: #5365, pull request 5369
- Change stats functions to always return lowercase names (Robin Geuze). ¶ References: #5287, pull request 5383
- Only use TCP Fast Open when supported and prevent compiler warnings. ¶ References: pull request 5449, pull request 5454
- Skip timeouts on the response latency graph. ¶ References: #5559, pull request 5563
- Copy the DNS header before encrypting it in place. ¶ References: #5566, pull request 5580

17.1.4 Removals

- Remove BlockFilter. ¶ References: #5513, pull request 5514
- Deprecate syntactic sugar functions. ¶ References: #5069, pull request 5526

17.1.5 misc

- Fix potential pointer wrap-around on 32 bits. ¶ References: pull request 5630
- Make the API available with an API key only. ¶ References: pull request 5631

17.2 1.1.0

Released December 29th 2016

Changes since 1.1.0-beta2:

17.2.1 Improvements

- #4783: Add -latomic on powerpc
- #4812: Handle header-only responses, handle Refused as Servfail in the cache

17.2.2 Bug fixes

- #4762: SuffixMatchNode: Fix an insertion issue for an existing node
- #4772: Fix dnsdist initscript config check

17.3 1.1.0-beta2

Released December 14th 2016

Changes since 1.1.0-beta1:

17.3.1 New features

- #4518: Fix dynblocks over TCP, allow refusing dyn blocked queries
- #4519: Allow altering the ECS behavior via rules and Lua
- #4535: Add `DNSQuestion:getDO()`
- #4653: `getStatisticsCounters()` to access counters from Lua
- #4657: Add `includeDirectory(dir)`
- #4658: Allow editing the ACL via the API
- #4702: Add `setUDPTimeout(n)`
- #4726: Add an option to return ServFail when no server is available
- #4748: Add `setCacheCleaningPercentage()`

17.3.2 Improvements

- #4533: Fix building with clang on OS X and FreeBSD
- #4537: Replace luawrapper's std::forward/std::make_tuple combo with std::forward_as_tuple (Sangwhan "fish" Moon)
- #4596: Change the default max number of queued TCP conns to 1000
- #4632: Improve dnsdist error message on a common typo/config mistake
- #4694: Don't use a const_iterator for erasing (fix compilation with some versions of gcc)
- #4715: Specify that dnsmessage.proto uses protobuf version 2
- #4765: Some service improvements

17.3.3 Bug fixes

- #4425: Fix a protobuf regression (requestor/responder mix-up) caused by a94673e
- #4541: Fix insertion issues in SuffixMatchTree, move it to dnsname.hh
- #4553: Flush output in single command client mode
- #4578: Fix destination address reporting
- #4640: Don't exit dnsdist on an exception in maintenance
- #4721: Handle exceptions in the UDP responder thread
- #4734: Add the TCP socket to the map only if the connection succeeds. Closes #4733
- #4742: Decrement the queued TCP conn count if writing to the pipe fails
- #4743: Ignore newBPFFilter() and newDynBPFFilter() in client mode
- #4753: Fix FD leak on TCP connection failure, handle TCP worker creation failure
- #4764: Prevent race while creating new TCP worker threads

17.4 1.1.0-beta1

Released September 1st 2016

Changes since 1.0.0:

17.4.1 New features

- #3762 Teeaction: send copy of query to second nameserver, sponge responses
- #3876 Add `showResponseRules()`, `{mv,rm,top}ResponseRule()`
- #3936 Filter on opcode, records count/type, trailing data
- #3975 Make dnsdist {A,I}XFR aware, document possible issues
- #4006 Add eBPF source address and qname/qtype filtering
- #4008 Node infrastructure for querying recent traffic
- #4042 Add server-side TCP Fast Open support
- #4050 Add `clearRules()` and `setRules()`
- #4114 Add `QNameLabelsCountRule()` and `QNameWireLengthRule()`
- #4116 Added src boolean to NetmaskGroupRule to match destination address (Reinier Schoof)
- #4175 Implemented query counting (Reinier Schoof)
- #4244 Add a `setCD` parameter to set cd=1 on health check queries
- #4284 Add RCodeRule(), Allow, Delay and Drop response actions
- #4305 Add an optional Lua callback for altering a Protobuf message
- #4309 Add showTCPStats function (RobinGeuze)
- #4329 Add options to LogAction() so it can append (instead of truncate) (Duane Wessels)

17.4.2 Improvements

- #3714 Add documentation links to dnsdist.service (Ruben Kerkhof)
- #3754 Allow the use of custom headers in the web server
- #3826 Implement a 'quiet' mode for SuffixMatchNodeRule()
- #3836 Log the content of webserver's exceptions
- #3858 Only log YaHTTP's parser exceptions in verbose mode
- #3877 Increase max FDs in systemd unit, warn if clearly too low
- #4019 Add an optional `addECS` option to `TeeAction()`
- #4029 Add version and feature information to version output
- #4079 Return an error on RemoteLog{,Response}Action() w/o protobuf
- #4246 API now sends pools as a JSON array instead of a string
- #4302 Add `help()` and `showVersion()`
- #4286 Add response rules to the API and Web status page
- #4068 Display the dyn eBPF filters stats in the web interface

17.4.3 Bug fixes

- #3755 Fix RegexRule example in dnsdistconf.lua
- #3773 Stop copying the HTTP request headers to the response
- #3837 Remove dnsdist service file on trusty
- #3840 Catch WrongTypeException in client mode
- #3906 Keep the servers ordered inside pools
- #3988 Fix `grepq()` output in the README
- #3992 Fix some typos in the AXFR/IXFR documentation
- #3995 Fix comparison between signed and unsigned integer
- #4049 Fix dnsdist rpm building script #4048 (Daniel Stirnimann)
- #4065 Include editline/readline.h instead of readline.h/history.h
- #4067 Disable eBPF support when BPF_FUNC_tail_call is not found
- #4069 Fix a buffer overflow when displaying an OpcodeRule
- #4101 Fix $ expansion in build-dnsdist-rpm
- #4198 newServer setting maxCheckFailures makes no sense (stutiredboy)
- #4205 Prevent the use of "any" addresses for downstream server
- #4220 Don't log an error when parsing an invalid UDP query
- #4348 Fix invalid outstanding count for {A,I}XFR over TCP
- #4365 Reset origFD asap to keep the outstanding count correct
- #4375 Tuple requires make_tuple to initialize
- #4380 Fix compilation with clang when eBPF support is enabled

17.5 1.0.0

Released April 21st 2016

Changes since 1.0.0-beta1:

17.5.1 Improvements

- #3700 Create user from the RPM package to drop privs
- #3712 Make check should run testrunner
- #3713 Remove contrib/dnsdist.service (Ruben Kerkhof)
- #3722 Use LT_INIT and disable static objects (Ruben Kerkhof)
- #3724 Include PDNS_CHECK_OS in configure (Christian Hofstaedtler)
- #3728 Document libedit Ctrl-R workaround for CentOS 6
- #3730 Make `topBandwidth()` behave like other top* functions
- #3731 Clarify a bit the documentation of load-balancing policies

17.5.2 Bug fixes

- #3711 Building rpm needs systemd headers (Ruben Kerkhof)
- #3736 Add missing Lua binding for NetmaskGroupRule()
- #3739 Drop privileges after daemonizing and writing our pid

17.6 1.0.0-beta1

Released April 14th 2016

Changes since 1.0.0-alpha2:

17.6.1 New features

- Per-pool packet cache
- Some actions do not stop the processing anymore when they match, allowing more complex setups: Delay, Disable Validation, Log, MacAddr, No Recurse and of course None
- The new RE2Rule() is available, using the RE2 regular expression library to match queries, in addition to the existing POSIX-based RegexRule()
- SpoofAction() now supports multiple A and AAAA records
- Remote logging of questions and answers via Protocol Buffer

17.6.2 Improvements

- #3405 Add health check logging, `maxCheckFailures` to backend
- #3412 Check config
- #3440 Client operation improvements
- #3466 Add dq binding for skipping packet cache in LuaAction (Jan Broer)
- #3499 Add support for multiple carbon servers
- #3504 Allow accessing the API with an optional API key
- #3556 Add an option to limit the number of queued TCP connections
- #3578 Add a `disable-syslog` option
- #3608 Export cache stats to carbon
- #3622 Display the ACL content on startup
- #3627 Remove ECS option from response's OPT RR when necessary
- #3633 Count "TTL too short" cache events
- #3677 systemd-notify support

17.6.3 Bug fixes

- #3388 Lock the Lua context before executing a LuaAction
- #3433 Check that the answer matches the initial query
- #3461 Fix crash when calling rmServer() with an invalid index
- #3550,#3551 Fix build failure on FreeBSD (Ruben Kerkhof)

- #3594 Prevent EOF error for empty console response w/o sodium
- #3634 Prevent dangling TCP fd in case setupTCPDownstream() fails
- #3641 Under threshold, QPS action should return None, not Allow
- #3658 Fix a race condition in MaxQPSIPRule

17.7 1.0.0-alpha2

Released February 5th 2016

Changes since 1.0.0-alpha1:

17.7.1 New features

- Lua functions now receive a DNSQuestion `dq` object instead of several parameters. This adds a greater compatibility with PowerDNS and allows adding more parameters without breaking the API (#3198)
- Added a `source` option to `newServer()` to specify the local address or interface used to contact a downstream server (#3138)
- CNAME and IPv6-only support have been added to spoofed responses (#3064)
- `grepq()` can be used to search for slow queries, along with `topSlow()`
- New Lua functions: `addDomainCNAMESpoof()`, `AllowAction()` by @bearggg, `exceedQRate()`,`MacAddrAction()`,`makeRule()`,`NotRule()`,`OrRule()`,`QClassRule()`, `RCodeAction()`, `SpoofCNAMEAction()`, `SuffixMatchNodeRule()`, `TCPRule()`, `topSlow()`
- `NetmaskGroup` support have been added in Lua (#3144)
- Added `MacAddrAction()` to add the source MAC address to the forwarded query (#3313)

17.7.2 Bug fixes

- An issue in DelayPipe could make dnsdist crash at startup
- `downstream-timeouts` metric was not always updated
- `truncateTC` was unproperly updating the response length (#3126)
- DNSCrypt responses larger than queries were unproperly truncated
- An issue prevented info message from being displayed in non-verbose mode, fixed by Jan Broer
- Reinstating an expired Dynamic Rule was not correctly logged (#3323)
- Initialized counters in the TCP client thread might have cause FD and memory leak, reported by Martin Pels (#3300)
- We now drop queries containing no question (qdcount == 0) (#3290)
- Outstanding TCP queries count was not always correct (#3288)
- A locking issue in exceedRespGen() might have caused crashs (#3277)
- Useless sockets were created in client mode (#3257)
- `addAnyTCRule()` was generating TC=1 responses even over TCP (#3251)

17.7.3 Web interface

- Cleanup of the HTML by Sander Hoentjen
- Fixed an XSS reported by @janeczku (#3217)
- Removed remote images
- Set the charset to UTF-8, added some security-related and CORS HTTP headers
- Added server latency by Jan Broer (#3201)
- Switched to official minified versions of JS scripts, by Sander Hoentjen (#3317)
- Don't log unauthenticated HTTP request as an authentication failure

17.7.4 Various documentation updates and minor cleanups:

- Added documentation for Advanced DNS Protection features (Dynamic rules, `maintenance()`)
- Make `topBandwidth()` default to the top 10 clients
- Replaced readline with libedit
- Added GPL2 License (#3200)
- Added incbin License (#3269)
- Updated completion rules
- Removed wrong option `--daemon-no` by Stefan Schmidt

17.8 1.0.0-alpha1

Released December 24th 2015

Initial release

CHAPTER

EIGHTEEN

UPGRADE GUIDE

18.1 1.1.0 to 1.2.0

In 1.2.0, several configuration options have been changed:

As the amount of possible settings for listen sockets is growing, all listen-related options must now be passed as a table as the second argument to both *addLocal()* and *setLocal()*. See the function's reference for more information.

The `BlockFilter` function is removed, as `addRule()` combined with a *DropAction()* can do the same.

CHAPTER

NINETEEN

SECURITY ADVISORIES

All security advisories for the DNSDist are listed here.

19.1 PowerDNS Security Advisory 2017-01 for dnsdist: Crafted backend responses can cause a denial of service

- CVE: CVE-2016-7069
- Date: 2017-08-21
- Credit: Guido Vranken
- Affects: dnsdist up to and including 1.2.0 on 32-bit systems
- Not affected: dnsdist 1.2.0, dnsdist on 64-bit (all versions)
- Severity: Low
- Impact: Degraded service or Denial of service
- Exploit: This issue can be triggered by sending specially crafted response packets from a backend
- Risk of system compromise: No
- Solution: Upgrade to a non-affected version
- Workaround: Disable EDNS Client Subnet addition

An issue has been found in dnsdist in the way EDNS0 OPT records are handled when parsing responses from a backend. When dnsdist is configured to add EDNS Client Subnet to a query, the response may contain an EDNS0 OPT record that has to be removed before forwarding the response to the initial client. On a 32-bit system, the pointer arithmetic used when parsing the received response to remove that record might trigger an undefined behavior leading to a crash.

dnsdist up to and including 1.1.0 is affected on 32-bit systems. dnsdist 1.2.0 is not affected, dnsdist on 64-bit systems is not affected.

For those unable to upgrade to a new version, a minimal patch is available for 1.1.0

We would like to thank Guido Vranken for finding and subsequently reporting this issue.

19.2 PowerDNS Security Advisory 2017-02 for dnsdist: Alteration of ACLs via API authentication bypass

- CVE: CVE-2017-7557
- Date: 2017-08-21
- Credit: Nixu

- Affects: dnsdist 1.1.0
- Not affected: dnsdist 1.0.0, 1.2.0
- Severity: Low
- Impact: Access restriction bypass
- Exploit: This issue can be triggered by tricking an authenticated user into visiting a crafted website
- Risk of system compromise: No
- Solution: Upgrade to a non-affected version
- Workaround: Keep the API read-only (default) via setAPIWritable(false)

An issue has been found in dnsdist 1.1.0, in the API authentication mechanism. API methods should only be available to a user authenticated via an X-API-Key HTTP header, and not to a user authenticated on the webserver via Basic Authentication, but it was discovered by Nixu during a source code audit that dnsdist 1.1.0 allows access to all API methods to both kind of users.

In the default configuration, the API does not provide access to more information than the webserver does, and therefore this issue has no security implication. However if the API is allowed to make configuration changes, via the setAPIWritable(true) option, this allows a remote unauthenticated user to trick an authenticated user into editing dnsdist's ACLs by making him visit a crafted website containing a Cross-Site Request Forgery.

For those unable to upgrade to a new version, a minimal patch is available for 1.1.0

CHAPTER

TWENTY

GLOSSARY

ACL Access Control List

Open Resolver A recursive DNS server available for many hosts on the internet. Ususally without adequate rate-limiting, allowing it to be used in reflection attacks.

QPS Queries Per Second

HTTP ROUTING TABLE

/api

/jsonstat

INDEX

E

G

I

J

L

M

N

O

P

Q

R

S

T

U

W