Knot Resolver

Release 5.0.0

CZ.NIC Labs

Jan 28, 2020

Quick Start

Knot Resolver is a minimalistic implementation of a caching validating DNS resolver. Modular architecture keeps the core tiny and efficient, and it provides a state-machine like API for extensions. Welcome to Knot Resolver Quick Start Guide! This chapter will guide you through first installation and basic setup recommended for your use-case.

Before we start let us explain basic conventions used in this text:

This is Linux/Unix shell command to be executed and an output from this command:

```
$ echo "This is output!"
This is output!
$ echo "We use sudo to execute commands as root:"
We use sudo to execute commands as root:
$ sudo id
uid=0(root) gid=0(root) groups=0(root)
```

Snippets from Knot Resolver's configuration file **do not start with $ sign** and look like this:

```
-- this is a comment
-- following line will start listening on IP address 192.0.2.1 port 53
net.listen('192.0.2.1')
```

CHAPTER 1

Installation

As a first step, configure your system to use upstream repositories which have the **latest version** of Knot Resolver. Follow the instructions below for your distribution.

Debian/Ubuntu

Note: Please note that the packages available in distribution repositories of Debian and Ubuntu are outdated. Make sure to follow these steps to use our upstream repositories.

```
$ wget https://secure.nic.cz/files/knot-resolver/knot-resolver-release.deb
$ sudo dpkg -i knot-resolver-release.deb
$ sudo apt update
$ sudo apt install -y knot-resolver
```

CentOS 7

```
$ sudo yum install -y https://secure.nic.cz/files/knot-resolver/knot-resolver-release.
↪el.rpm
$ sudo yum install -y knot-resolver
```

Fedora

```
$ sudo dnf install -y https://secure.nic.cz/files/knot-resolver/knot-resolver-release.
↪fedora.rpm
$ sudo dnf install -y knot-resolver
```

Arch Linux

Use knot-resolver package from AUR.

openSUSE Leap / Tumbleweed Add the OBS package repository home:CZ-NIC:knot-resolver-latest to your system.

CHAPTER 2

Startup

The simplest way to run single instance of Knot Resolver is to use provided Knot Resolver's Systemd integration:

```
$ sudo systemctl start kresd@1.service
```

See logs and status of running instance with `systemctl status kresd@1.service` command. For more information about Systemd integration see `man kresd.systemd`.

Warning: `kresd@*.service` is not enabled by default, thus Knot Resolver won't start automatically after reboot. To start and enable service in one command use `systemctl enable --now kresd@1.service`

2.1 First DNS query

After installation and first startup, Knot Resolver's default configuration accepts queries on loopback interface. This allows you to test that the installation and service startup were successful before continuing with configuration.

For instance, you can use DNS lookup utility `kdig` to send DNS queries. The `kdig` command is provided by following packages:

Distribution	package with kdig
Arch	knot
CentOS	knot-utils
Debian	knot-dnsutils
Fedora	knot-utils
OpenSUSE	knot-utils
Ubuntu	knot-dnsutils

The following query should return list of Root Name Servers:

```
$ kdig +short @localhost . NS
a.root-servers.net.
...
m.root-servers.net.
```

CHAPTER 3

Configuration

Note: When copy&pasting examples from this manual please pay close attention to brackets and also line ordering - order of lines matters.

The configuration language is in fact Lua script, so you can use full power of this programming language. See article Learn Lua in 15 minutes for a syntax overview.

Easiest way to configure Knot Resolver is to paste your configuration into configuration file `/etc/knot-resolver/kresd.conf`. Complete configurations files for examples in this chapter can be found here. The example configuration files are also installed as documentation files, typically in directory `/usr/share/doc/knot-resolver/examples/` (their location may be different based on your Linux distribution). Detailed configuration of daemon and implemented modules can be found in configuration reference:

3.1 Listening on network interfaces

Network interfaces to listen on and supported protocols are configured using *`net.listen()`* function.

The following configuration instructs Knot Resolver to receive standard unencrypted DNS queries on IP addresses *192.0.2.1* and *2001:db8::1*. Encrypted DNS queries are accepted using DNS-over-TLS protocol on all IP addresses configured on network interface *eth0*, TCP port 853.

```
-- unencrypted DNS on port 53 is default
net.listen('192.0.2.1')
net.listen('2001:db8::1')
net.listen(net.eth0, 853, { kind = 'tls' })
```

Warning: On machines with multiple IP addresses on the same interface avoid listening on wildcards `0.0.0.0` or `::`. Knot Resolver could answer from different IP addresses if the network address ranges overlap, and clients would refuse such a response.

3.2 Scenario: Internal Resolver

This is an example of typical configuration for company-internal resolver which is not accessible from outside of company network.

3.2.1 Internal-only domains

An internal-only domain is a domain not accessible from the public Internet. In order to resolve internal-only domains a query policy has to be added to forward queries to a correct internal server. This configuration will forward two listed domains to a DNS server with IP address `192.0.2.44`.

```
-- define list of internal-only domains
internalDomains = policy.todnames({'company.example', 'internal.example'})

-- forward all queries belonging to domains in the list above to IP address '192.0.2.
↪44'
policy.add(policy.suffix(policy.FORWARD({'192.0.2.44'}), internalDomains))
```

3.3 Scenario: ISP Resolver

The following configuration is typical for Internet Service Providers who offer DNS resolver service to their own clients in their own network. Please note that running a *public DNS resolver* is more complicated and not covered by this quick start guide.

3.3.1 Limiting client access

With exception of public resolvers, a DNS resolver should resolve only queries sent by clients in its own network. This restriction limits attack surface on the resolver itself and also for the rest of the Internet.

In a situation where access to DNS resolver is not limited using IP firewall, you can implement access restrictions using the *view module* which combines query source information with *policy rules*. Following configuration allows only queries from clients in subnet 192.0.2.0/24 and refuses all the rest.

```
modules.load('view')

-- whitelist queries identified by subnet
view:addr('192.0.2.0/24', policy.all(policy.PASS))
```

(continues on next page)

(continued from previous page)

```
-- drop everything that hasn't matched
view:addr('0.0.0.0/0', policy.all(policy.DROP))
```

3.3.2 TLS server configuration

Today clients are demanding secure transport for DNS queries between client machine and DNS resolver. The recommended way to achieve this is to start DNS-over-TLS server and accept also encrypted queries.

First step is to enable TLS on listening interfaces:

```
net.listen('192.0.2.1', 853, { kind = 'tls' })
net.listen('2001::db8:1', 853, { kind = 'tls' })
```

By default a self-signed certificate is generated. Second step is then obtaining and configuring your own TLS certificates signed by a trusted CA. Once the certificate was obtained a path to certificate files can be specified using function *net.tls()*:

```
net.tls("/etc/knot-resolver/server-cert.pem", "/etc/knot-resolver/server-key.pem")
```

3.3.3 Mandatory domain blocking

Some jurisdictions mandate blocking access to certain domains. This can be achieved using following *policy rule*:

```
policy.add(
      policy.suffix(policy.DENY,
              policy.todnames({'example.com.', 'blocked.example.net.'})))
```

3.4 Scenario: Personal Resolver

DNS queries can be used to gather data about user behavior. Knot Resolver can be configured to forward DNS queries elsewhere, and to protect them from eavesdropping by TLS encryption.

Warning: Latest research has proven that encrypting DNS traffic is not sufficient to protect privacy of users. For this reason we recommend all users to use full VPN instead of encrypting *just* DNS queries. Following configuration is provided **only for users who cannot encrypt all their traffic**. For more information please see following articles:

- Simran Patil and Nikita Borisov. 2019. What can you learn from an IP? (slides, the article itself)
- Bert Hubert. 2019. Centralised DoH is bad for Privacy, in 2019 and beyond

3.4.1 Forwarding over TLS protocol (DNS-over-TLS)

Forwarding over TLS protocol protects DNS queries sent out by resolver. It can be configured using *policy.TLS_FORWARD* function which provides methods for authentication. See list of DNS Privacy Test Servers supporting DNS-over-TLS to test your configuration.

Read more on *Forwarding over TLS protocol (DNS-over-TLS)*.

3.4.2 Forwarding to multiple targets

With the use of *policy.slice* function, it is possible to split the entire DNS namespace into distinct "slices". When used in conjunction with *policy.TLS_FORWARD*, it's possible to forward different queries to different remote resolvers. As a result no single remote resolver will get complete list of all queries performed by this client.

Warning: Beware that this method has not been scientifically tested and there might be types of attacks which will allow remote resolvers to infer more information about the client. Again: If possible encypt **all** your traffic and not just DNS queries!

```
policy.add(policy.slice(
   policy.slice_randomize_psl(),
   policy.TLS_FORWARD({{'192.0.2.1', hostname='res.example.com'}}),
   policy.TLS_FORWARD({
      -- multiple servers can be specified for a single slice
      -- the one with lowest round-trip time will be used
      {'193.17.47.1', hostname='odvr.nic.cz'},
      {'185.43.135.1', hostname='odvr.nic.cz'},
   })
))
```

3.4.3 Non-persistent cache

Knot Resolver's cache contains data clients queried for. If you are concerned about attackers who are able to get access to your computer system in power-off state and your storage device is not secured by encryption you can move the cache to tmpfs. See chapter *Persistence*.

Congratulations! Your resolver is now up and running and ready for queries. For serious deployments do not forget to read *Configuration* and *Operation* chapters.

CHAPTER 4

Configuration Overview

Configuration file is named `/etc/knot-resolver/kresd.conf` and is read when you execute Knot Resolver using systemd commands described in section *Startup*.[1]

4.1 Syntax

The configuration file syntax allows you to specify different kinds of data:

- `group.option = 123456`
- `group.option = "string value"`
- `group.command(123456, "string value")`
- `group.command({ key1 = "value1", key2 = 222, key3 = "third value" })`
- `globalcommand(a_parameter_1, a_parameter_2, a_parameter_3, etc)`
- `-- any text after -- sign is ignored till end of line`

Following **configuration file snippet** starts listening for unencrypted and also encrypted DNS queries on IP address 192.0.2.1, and sets cache size.

```
-- this is a comment: listen for unencrypted queries
net.listen('192.0.2.1')
-- another comment: listen for queries encrypted using TLS on port 853
net.listen('192.0.2.1', 853, { kind = 'tls' })
-- 10 MB cache is suitable for a very small deployment
cache.size = 10 * MB
```

Tip: When copy&pasting examples from this manual please pay close attention to brackets and also line ordering - order of lines matters.

[1] If you decide to run binary `/usr/sbin/kresd` manually (instead of using systemd) do not forget to specify `-c` option with path to configuration file, otherwise `kresd` will read file named `config` from its current working directory.

The configuration language is in fact Lua script, so you can use full power of this programming language. See article Learn Lua in 15 minutes for a syntax overview.

When you modify configuration file on disk restart resolver process to get changes into effect. See chapter *Zero-downtime restarts* if even short outages are not acceptable for your deployment.

4.2 Documentation Conventions

Besides text configuration file, Knot Resolver also supports interactive and dynamic configuration using scripts or external systems, which is described in chapter *Run-time reconfiguration*. Through this manual we present examples for both usage types - static configuration in a text file (see above) and also the interactive mode.

The **interactive prompt** is denoted by >, so all examples starting with > character are transcripts of user (or script) interaction with Knot Resolver and resolver's responses. For example:

```
> -- this is a comment entered into interactive prompt
> -- comments have no effect here
> -- the next line shows a command entered interactivelly and its output
> verbose()
false
> -- the previous line without > character is output from verbose() command
```

Following example demontrates how to interactivelly list all currently loaded modules, and includes multi-line output:

```
> modules.list()
[1] => iterate
[2] => validate
[3] => cache
```

Before we dive into configuring features, let us explain modularization basics.

4.3 Modules

Knot Resolver functionality consists of separate modules, which allow you to mix-and-match features you need without slowing down operation by features you do not use.

This practically means that you need to load module before using features contained in it, for example:

```
-- load module and make dnstap features available
modules.load('dnstap')
-- configure dnstap features
dnstap.config({
        socket_path = "/tmp/dnstap.sock"
})
```

Obviously ordering matters, so you have to load module first and configure it after it is loaded.

Here is full reference manual for module configuration:

modules.list()

Returns List of loaded modules.

modules.load(name)

Parameters

- **name** (*string*) – Module name, e.g. “hints”

Returns `true` if modules was (or already is) loaded, error otherwise.

Load a module by name.

modules.unload(name)

Parameters

- **name** (*string*) – Module name, e.g. “detect_time_jump”

Returns `true` if modules was unloaded, error otherwise.

Unload a module by name. This is useful for unloading modules loaded by default, mainly for debugging purposes.

Now you know what configuration file to modify, how to read examples and what modules are so you are ready for a real configuration work!

CHAPTER 5

Networking and protocols

This section describes configuration of network interfaces and protocols. Please keep in mind that DNS resolvers act as *DNS server* and *DNS client* at the same time, and that these roles require different configuration.

This picture illustrates different actors involved DNS resolution process, supported protocols, and clarifies what we call *server configuration* and *client configuration*.

Attribution: Icons by Bernar Novalyi from the Noun Project

For *resolver's clients* the resolver itself acts as a DNS server.

After receiving a query the resolver will attempt to find answer in its cache. If the data requested by resolver's client is not available in resolver's cache (so-called *cache-miss*) the resolver will attempt to obtain the data from servers *upstream* (closer to the source of information), so at this point the resolver itself acts like a DNS client and will send DNS query to other servers.

By default the Knot Resolver works in recursive mode, i.e. the resolver will contact authoritative servers on the Internet. Optionally it can be configured in forwarding mode, where cache-miss queries are *forwarded to another DNS resolver* for processing.

5.1 Server (communication with clients)

5.1.1 Addresses and services

Addresses, ports, protocols, and API calls available for clients communicating with resolver are configured using `net.listen()`.

First you need to decide what service should be available on given IP address + port combination.

Protocol/service	net.listen *kind*
DNS (unencrypted UDP+TCP, **RFC 1034**)	`dns`
DNS-over-TLS (DoT)	`tls`
DNS-over-HTTP (DoH)	`doh`
Web management	`webmgmt`
Control socket	`control`

Note: By default, **unencrypted DNS and DNS-over-TLS** are configured to **listen on localhost**.

Control sockets are created either in `/run/knot-resolver/control/` (when using systemd) or `$PWD/control/`.

`net.listen` (addresses, [port = *53*, { kind = *'dns'*, freebind = false *}*])

Returns `true` if port is bound, an error otherwise

Listen on addresses; port and flags are optional. The addresses can be specified as a string or device. Port 853 implies `kind = 'tls'` but it is always better to be explicit. Freebind allows binding to a non-local or not yet available address.

Network protocol	Configuration command
DNS (UDP+TCP, **RFC 1034**)	`net.listen('192.0.2.123', 53)`
DNS-over-TLS (DoT)	`net.listen('192.0.2.123', 853, { kind = 'tls' })`
DNS-over-HTTP (DoH)	`net.listen('192.0.2.123', 443, { kind = 'doh' })`
Web management	`net.listen('192.0.2.123', 8453, { kind = 'webmgmt' })`
Control socket	`net.listen('/tmp/kres.control', nil, { kind = 'control' })`

Examples:

```
net.listen('::1')
net.listen(net.lo, 53)
net.listen(net.eth0, 853, { kind = 'tls' })
net.listen('192.0.2.1', 53, { freebind = true })
net.listen({'127.0.0.1', '::1'}, 53, { kind = 'dns' })
net.listen('::', 443, { kind = 'doh' }) -- see http module
net.listen('::', 8453, { kind = 'webmgmt' }) -- see http module
net.listen('/tmp/kresd-socket', nil, { kind = 'webmgmt' }) -- http module␣
↪supports AF_UNIX
```

Warning: Make sure you read section *DNS-over-HTTP (DoH)* before exposing the DNS-over-HTTP protocol to outside.

Warning: On machines with multiple IP addresses avoid listening on wildcards `0.0.0.0` or `::`. Knot Resolver could answer from different IP addresses if the network address ranges overlap, and clients would probably refuse such a response.

Features for scripting

Following configuration functions are useful mainly for scripting or *Run-time reconfiguration*.

net.close(address, [port])

Returns boolean (at least one endpoint closed)

Close all endpoints listening on the specified address, optionally restricted by port as well.

net.list()

Returns Table of bound interfaces.

Example output:

```
[1] => {
    [kind] => tls
    [transport] => {
        [family] => inet4
        [ip] => 127.0.0.1
        [port] => 853
        [protocol] => tcp
    }
}
[2] => {
    [kind] => dns
    [transport] => {
        [family] => inet6
        [ip] => ::1
        [port] => 53
        [protocol] => udp
    }
}
[3] => {
    [kind] => dns
    [transport] => {
        [family] => inet6
        [ip] => ::1
        [port] => 53
        [protocol] => tcp
    }
}
```

net.interfaces()

Returns Table of available interfaces and their addresses.

Example output:

```
[lo0] => {
    [addr] => {
        [1] => ::1
        [2] => 127.0.0.1
    }
    [mac] => 00:00:00:00:00:00
}
[eth0] => {
    [addr] => {
        [1] => 192.168.0.1
    }
```

(continues on next page)

(continued from previous page)

```
    [mac] => de:ad:be:ef:aa:bb
}
```

Tip: You can use `net.<iface>` as a shortcut for specific interface, e.g. `net.eth0`

net.tcp_pipeline ([len])
Get/set per-client TCP pipeline limit, i.e. the number of outstanding queries that a single client connection can make in parallel. Default is 100.

```
> net.tcp_pipeline()
100
> net.tcp_pipeline(50)
50
```

Warning: Please note that too large limit may have negative impact on performance and can lead to increased number of SERVFAIL answers.

5.1.2 DNS-over-TLS server (DoT)

DNS-over-TLS server (**RFC 7858**) is enabled by default on localhost. Information how to configure listening on specific IP addresses is in previous sections: *net.listen()*.

By default a self-signed certificate is generated. For serious deployments it is strongly recommended to configure your own TLS certificates signed by a trusted CA. This is done using function *net.tls()*.

net.tls ([cert_path], [key_path])
Get/set path to a server TLS certificate and private key for DNS/TLS.

Example output:

```
> net.tls("/etc/knot-resolver/server-cert.pem", "/etc/knot-resolver/server-key.pem
↪")
> net.tls()  -- print configured paths
("/etc/knot-resolver/server-cert.pem", "/etc/knot-resolver/server-key.pem")
```

net.tls_padding ([true | *false*])
Get/set EDNS(0) padding of answers to queries that arrive over TLS transport. If set to *true* (the default), it will use a sensible default padding scheme, as implemented by libknot if available at compile time. If set to a numeric value >= 2 it will pad the answers to nearest *padding* boundary, e.g. if set to *64*, the answer will have size of a multiple of 64 (64, 128, 192, ...). If set to *false* (or a number < 2), it will disable padding entirely.

net.tls_sticket_secret ([string with pre-shared *secret*])
Set secret for TLS session resumption via tickets, by **RFC 5077**.

The server-side key is rotated roughly once per hour. By default or if called without secret, the key is random. That is good for long-term forward secrecy, but multiple kresd instances won't be able to resume each other's sessions.

If you provide the same secret to multiple instances, they will be able to resume each other's sessions *without* any further communication between them. This synchronization works only among instances having the same endianess and time_t structure and size (*sizeof(time_t)*).

For good security the secret must have enough entropy to be hard to guess, and it should still be occasionally rotated manually and securely forgotten, to reduce the scope of privacy leak in case the secret leaks eventually.

> **Warning: Setting the secret is probably too risky with TLS <= 1.2**. GnuTLS stable release supports TLS 1.3 since 3.6.3 (summer 2018). Therefore setting the secrets should be considered experimental for now and might not be available on your system.

net.tls_sticket_secret_file([string with path to a file containing pre-shared *secret]*)
The same as *net.tls_sticket_secret()*, except the secret is read from a (binary) file.

5.1.3 HTTP services

Tip: In most distributions, the `http` module is available from a separate package `knot-resolver-module-http`. The module isn't packaged for openSUSE.

This module does the heavy lifting to provide an HTTP and HTTP/2 enabled server which provides few built-in services and also allows other modules to export restful APIs and websocket streams.

One example is statistics module that can stream live metrics on the website, or publish metrics on request for Prometheus scraper, and also *DNS-over-HTTP (DoH)*.

By default this module provides two kinds of endpoints, and unlimited number of "used-defined kinds" can be added in configuration.

Endpoint	Explanation
doh	*DNS-over-HTTP (DoH)*
webmgmt	*built-in web management* APIs (includes DoH)

Each network address and port combination can be configured to expose one kind of endpoint. This is done using the same mechanisms as network configuration for plain DNS and DNS-over-TLS, see chapter *Networking and protocols* for more details.

> **Warning:** Management endpoint (`webmgmt`) must not be directly exposed to untrusted parties. Use reverse-proxy like Apache or Nginx if you need to authenticate API clients for the management API.

By default all endpoints share the same configuration for TLS certificates etc. This can be changed using `http.config()` configuration call explained below.

Example configuration

This section shows how to configure HTTP module itself. For information how to configure HTTP server's IP addresses and ports please see chapter *Networking and protocols*.

```
-- load HTTP module with defaults (self-signed TLS cert)
modules.load('http')
-- optionally load geoIP database for server map
http.config({
        geoip = 'GeoLite2-City.mmdb',
```

(continues on next page)

(continued from previous page)

```
        -- e.g. https://dev.maxmind.com/geoip/geoip2/geolite2/
        -- and install mmdblua library
})
```

Now you can reach the web services and APIs, done!

```
$ curl -k https://localhost:8453
$ curl -k https://localhost:8453/stats
```

HTTPS (TLS for HTTP)

By default, the web interface starts HTTPS/2 on specified port using an ephemeral TLS certificate that is valid for 90 days and is automatically renewed. It is of course self-signed. Why not use something like Let's Encrypt?

Warning: If you use package `luaossl < 20181207`, intermediate certificate is not sent to clients, which may cause problems with validating the connection in some cases.

You can disable unecrypted HTTP and enforce HTTPS by passing `tls = true` option for all HTTP endpoints:

```
http.config({
        tls = true,
})
```

It is also possible to provide different configuration for each kind of endpoint, e.g. to enforce TLS and use custom certificate only for DoH:

```
http.config({
        tls = true,
        cert = '/etc/knot-resolver/mycert.crt',
        key  = '/etc/knot-resolver/mykey.key',
}, 'doh')
```

The format of both certificate and key is expected to be PEM, e.g. equivalent to the outputs of following:

```
openssl ecparam -genkey -name prime256v1 -out mykey.key
openssl req -new -key mykey.key -out csr.pem
openssl req -x509 -days 90 -key mykey.key -in csr.pem -out mycert.crt
```

It is also possible to disable HTTPS altogether by passing `tls = false` option. Plain HTTP gets handy if you want to use reverse-proxy like Apache or Nginx for authentication to API etc. (Unencrypted HTTP could be fine for localhost tests as, for example, Safari doesn't allow WebSockets over HTTPS with a self-signed certificate. Major drawback is that current browsers won't do HTTP/2 over insecure connection.)

Warning: If you use multiple Knot Resolver instances with these automatically maintained ephemeral certificates, they currently won't be shared. It's assumed that you don't want a self-signed certificate for serious deployments anyway.

Built-in services

The HTTP module has several built-in services to use.

Endpoint	Service	Description
`/stats`	Statistics/metrics	Exported *metrics* from *Statistics collector* in JSON format.
`/metrics`	Prometheus metrics	Exported metrics for Prometheus.
`/trace/:name/:type`	Tracking	*Trace resolution* of a DNS query and return the verbose logs.
`/doh`	DNS-over-HTTP	**RFC 8484** endpoint, see *DNS-over-HTTP (DoH)*.

Dependencies

- lua-http (>= 0.3) available in LuaRocks

  If you're installing via Homebrew on OS X, you need OpenSSL too.

  ```
  $ brew update
  $ brew install openssl
  $ brew link openssl --force # Override system OpenSSL
  ```

  Any other system can install from LuaRocks directly:

  ```
  $ luarocks install http
  ```

- (*optional*) mmdblua available in LuaRocks

  ```
  $ luarocks install --server=https://luarocks.org/dev mmdblua
  $ curl -O https://geolite.maxmind.com/download/geoip/database/GeoLite2-
  ↪City.mmdb.gz
  $ gzip -d GeoLite2-City.mmdb.gz
  ```

5.1.4 DNS-over-HTTP (DoH)

Warning:

- DoH support was added in version 4.0.0 and is subject to change.
- DoH implementation in Knot Resolver is intended for experimentation only as there is insufficient experience with the module and the DoH protocol in general.
- For the time being it is recommended to run DoH endpoint on a separate machine which is not handling normal DNS operations.
- More information about controversies around the DoH can be found in blog posts DNS Privacy at IETF 104 and More DOH by Geoff Huston.
- Knot Resolver developers do not endorse use of the DoH protocol.

Following section compares several options for running a DoH capable server. Make sure you read through this chapter before exposing the DoH service to users.

DoH support in Knot Resolver

The *HTTP module* in Knot Resolver also provides support for binary DNS-over-HTTP protocol standardized in **RFC 8484**.

This integrated DoH server has following properties:

Scenario HTTP module in Knot Resolver configured to provide `/doh` endpoint (as shown below).

Advantages

- Integrated solution provides management and monitoring in one place.
- Supports ACLs for DNS traffic based on client's IP address.

Disadvantages

- Exposes Knot Resolver instance to attacks over HTTP.
- Does not offer fine grained authorization and logging at HTTP level.
- Let's Encrypt integration is not automated.

Example configuration is part of examples for generic HTTP module. After configuring your endpoint you can reach the DoH endpoint using URL `https://your.resolver.hostname.example/doh`, done!

```
# query for www.knot-resolver.cz AAAA
$ curl -k https://your.resolver.hostname.example/doh?
↪dns=l1sBAAABAAAAAAAAA3d3dw1rbm90LXJlc29sdmVyAmN6AAAcAAE
```

Please see section *HTTPS (TLS for HTTP)* for further details about TLS configuration.

Alternative configurations use HTTP proxies between clients and a Knot Resolver instance:

Normal HTTP proxy

Scenario A standard HTTP-compliant proxy is configured to proxy *GET* and *POST* requests to HTTP endpoint */doh* to a machine running Knot Resolver.

Advantages

- Protects Knot Resolver instance from *some* types of attacks at HTTP level.
- Allows fine-grained filtering and logging at HTTP level.
- Let's Encrypt integration is readily available.
- Is based on mature software.

Disadvantages

- Fine-grained ACLs for DNS traffic are not available because proxy hides IP address of client sending DNS query.
- More complicated setup with two components (proxy + Knot Resolver).

HTTP proxy with DoH support

Scenario HTTP proxy extended with a special module for DNS-over-HTTP. The module transforms HTTP requests to standard DNS queries which are then processed by Knot Resolver. DNS replies from Knot Resolver are then transformed back to HTTP encoding by the proxy.

Advantages

- Protects Knot Resolver instance from *all* attacks at HTTP level.
- Allows fine-grained filtering and logging at HTTP level.
- Let's Encrypt integration is readily available if proxy is based on a standard HTTP software.

Disadvantages

- Fine-grained ACLs for DNS traffic are not available because proxy hides IP address of client sending DNS query. (Unless proxy and resolver are using non-standard packet extensions like DNS X-Proxied-For.)
- More complicated setup with three components (proxy + special module + Knot Resolver).

Client configuration

Most common client today is web browser Firefox. Relevant configuration is described e.g. in following article. To use your own DoH server just change `network.trr.uri` configuration option to match URL of your DoH endpoint.

More detailed description of configuration options in Firefox can be found in article Inside Firefox's DOH engine by Daniel Stenberg.

Warning: Please note that Knot Resolver developers are not as enthusiastic about DoH technology as author of the article linked above, make sure you read *warnings at beginning of this section*.

5.2 Client (retrieving answers from servers)

Following chapters describe basic configuration of how resolver retrieves data from other *(upstream)* servers. Data processing is also affected by configured policies, see chapter *Policy, access control, data manipulation* for more advanced usage.

5.2.1 IPv4 and IPv6 usage

Following settings affect client part of the resolver, i.e. communication between the resolver itself and other DNS servers.

IPv4 and IPv6 protocols are used by default. For performance reasons it is recommended to explicitly disable protocols which are not available on your system.

`net.ipv4 = true|false`

> **Return** boolean (default: true)

Enable/disable using IPv4 for contacting upstream nameservers.

`net.ipv6 = true|false`

> **Return** boolean (default: true)

Enable/disable using IPv6 for contacting upstream nameservers.

`net.outgoing_v4` ([string *address]*)
Get/set the IPv4 address used to perform queries. The default is `nil`, which lets the OS choose any address.

`net.outgoing_v6` ([string *address]*)
Get/set the IPv6 address used to perform queries. The default is `nil`, which lets the OS choose any address.

5.2.2 Forwarding

Forwarding configuration instructs resolver to forward cache-miss queries from clients to manually specified DNS resolvers *(upstream servers)*. In other words the *forwarding* mode does exact opposite of the default *recursive* mode because resolver in *recursive* mode automatically selects which servers to ask.

Main use-cases are:

- Building a tree structure of DNS resolvers to improve performance (by improving cache hit rate).
- Accessing domains which are not available using recursion (e.g. if internal company servers return different anusers than public ones).
- Forwarding through a central DNS traffic filter.

Forwarding implementation in Knot Resolver has following properties:

- Answers from *upstream* servers are cached.
- Answers from *upstream* servers are locally DNSSEC-validated, unless `policy.STUB` is used.
- Resolver automatically selects which IP address from given set of IP addresses will be used (based on performance characteristics).
- Forwarding can use either unencrypted DNS protocol, or *Forwarding over TLS protocol (DNS-over-TLS)*.

Warning: We strongly discourage use of "fake top-level domains" like `corp.` because these made-up domains are indistinguishable from an attack, so DNSSEC validation will prevent such domains from working. If you *really* need a variant of forwarding which does not DNSSEC-validate received data please see chapter *Replacing part of the DNS tree*. In long-term it is better to migrate data into a legitimate, properly delegated domains which do not suffer from these security problems.

Simple examples for **unencrypted** forwarding:

```
-- forward all traffic to specified IP addresses (selected automatically)
policy.add(policy.all(policy.FORWARD({'2001:db8::1', '192.0.2.1'})))

-- forward only queries for names under domain example.com to a single IP address
policy.add(policy.suffix(policy.FORWARD('192.0.2.1'), {todname('example.com.')}))
```

To configure encrypted version please see chapter *Forwarding over TLS protocol (DNS-over-TLS)*.

Forwarding is documented in depth together with rest of *Query policies*.

5.3 DNS protocol tweaks

5.3.1 DNS protocol tweaks

Following settings change low-level details of DNS protocol implementation. Default values should not be changed except for very special cases.

net.bufsize ([udp_bufsize])
Get/set maximum EDNS payload size advertised in DNS packets. Default is 4096 bytes and the default will be lowered to value around 1220 bytes in future, once DNS Flag Day 2020 becomes effective.

Minimal value allowed by standard **RFC 6891** is 512 bytes, which is equal to DNS packet size without Extension Mechanisms for DNS. Value 1220 bytes is minimum size required in DNSSEC standard **RFC 4035**.

Example output:

```
> net.bufsize(4096)
nil
```

(continues on next page)

(continued from previous page)

```
> net.bufsize()
4096
```

Module *workarounds* resolver behavior on specific broken sub-domains. Currently it mainly disables case randomization.

```
modules.load('workarounds < iterate')
```

CHAPTER 6

Performance and resiliency

For DNS resolvers, the most important parameter from performance perspective is cache hit rate, i.e. percentage of queries answered from resolver's cache. Generally the higher cache hit rate the better.

Performance tunning should start with cache *Sizing* and *Persistence*.

It is also recommended to run *Multiple instances* (even on a single machine!) because it allows to utilize multiple CPU threads and increases overall resiliency.

Other features described in this section can be used for fine-tunning performance and resiliency of the resolver but generally have much smaller impact than cache settings and number of instances.

6.1 Cache

Cache in Knot Resolver is stored on disk and also shared between *Multiple instances* so resolver doesn't lose the cached data on restart or crash.

To improve performance even further the resolver implements so-called aggressive caching for DNSSEC-validated data (**RFC 8198**), which improves performance and also protects against some types of Random Subdomain Attacks.

6.1.1 Sizing

For personal and small office use-cases cache size around 100 MB is more than enough.

For large deployments we recommend to run Knot Resolver on a dedicated machine, and to allocate 90% of machine's free memory for resolver's cache.

For example, imagine you have a machine with 16 GB of memory. After machine restart you use command `free -m` to determine amount of free memory (without swap):

```
$ free -m
              total        used        free
Mem:          15907         979       14928
```

Now you can configure cache size to be 90% of the free memory 14 928 MB, i.e. 13 453 MB:

```
-- 90 % of free memory after machine restart
cache.size = 13453 * MB
```

Note: The *Garbage Collector* can be used to periodically trim the cache. It is enabled and configured by default when running kresd with systemd integration.

6.1.2 Persistence

Tip: Using tmpfs for cache improves performance and reduces disk I/O.

By default the cache is saved on a persistent storage device so the content of the cache is persisted during system reboot. This usually leads to smaller latency after restart etc., however in certain situations a non-persistent cache storage might be preferred, e.g.:

- Resolver handles high volume of queries and I/O performance to disk is too low.
- Threat model includes attacker getting access to disk content in power-off state.
- Disk has limited number of writes (e.g. flash memory in routers).

If non-persistent cache is desired configure cache directory to be on tmpfs filesystem, a temporary in-memory file storage. The cache content will be saved in memory, and thus have faster access and will be lost on power-off or reboot.

Note: In most of the Unix-like systems `/tmp` and `/var/run` are commonly mounted to tmpfs. While it is technically possible to move the cache to an existing tmpfs filesystem, it is *not recommended*: The path to cache is specified in multiple systemd units, and a shared tmpfs space could be used up by other applications, leading to `SIGBUS` errors during runtime.

Mounting the cache directory as tmpfs is recommended approach. Make sure to use appropriate `size=` option and don't forget to adjust the size in the config file as well.

```
# /etc/fstab
tmpfs        /var/cache/knot-resolver        tmpfs   rw,size=2G,uid=knot-resolver,
↪gid=knot-resolver,nosuid,nodev,noexec,mode=0700 0 0
```

```
# /etc/knot-resolver/config
cache.size = 2 * GB
```

6.1.3 Configuration reference

cache.open (max_size[, config_uri])

Parameters

- **max_size** (*number*) – Maximum cache size in bytes.

Returns `true` if cache was opened

Open cache with a size limit. The cache will be reopened if already open. Note that the max_size cannot be lowered, only increased due to how cache is implemented.

Tip: Use `kB, MB, GB` constants as a multiplier, e.g. `100*MB`.

The URI `lmdb://path` allows you to change the cache directory.

Example:

```
cache.open(100 * MB, 'lmdb:///var/cache/knot-resolver')
```

cache.size

Set the cache maximum size in bytes. Note that this is only a hint to the backend, which may or may not respect it. See *cache.open()*.

```
cache.size = 100 * MB -- equivalent to `cache.open(100 * MB)`
```

cache.current_size

Get the maximum size in bytes.

```
print(cache.current_size)
```

cache.storage

Set the cache storage backend configuration, see *cache.backends()* for more information. If the new storage configuration is invalid, it is not set.

```
cache.storage = 'lmdb://.'
```

cache.current_storage

Get the storage backend configuration.

```
print(cache.current_storage)
```

cache.backends()

Returns map of backends

Note: For now there is only one backend implementation, even though the APIs are ready for different (synchronous) backends.

The cache supports runtime-changeable backends, using the optional **RFC 3986** URI, where the scheme represents backend protocol and the rest of the URI backend-specific configuration. By default, it is a `lmdb` backend in working directory, i.e. `lmdb://`.

Example output:

```
[lmdb://] => true
```

cache.count()

Returns Number of entries in the cache. Meaning of the number is an implementation detail and is subject of change.

cache.close()

Returns `true` if cache was closed

Close the cache.

Note: This may or may not clear the cache, depending on the cache backend.

cache.stats()
Return table with low-level statistics for each internal cache operation. This counts each access to cache and does not directly map to individual DNS queries or resource records. For query-level statistics see *stats module*.

Example:

```
> cache.stats()
[read_leq_miss] => 4
[write] => 189
[read_leq] => 9
[read] => 4313
[read_miss] => 1143
[open] => 0
[close] => 0
[remove_miss] => 0
[commit] => 117
[match_miss] => 2
[match] => 21
[count] => 2
[clear] => 0
[remove] => 17
```

Cache operation *read_leq* (*read less or equal*, i.e. range search) was requested 9 times, and 4 out of 9 operations were finished with *cache miss*.

cache.max_ttl([ttl])

Parameters

- **ttl** (*number*) – maximum cache TTL in seconds (default: 6 days)

Returns current maximum TTL

Get or set maximum cache TTL.

Note: The *ttl* value must be in range *(min_ttl, 4294967295)*.

Warning: This settings applies only to currently open cache, it will not persist if the cache is closed or reopened.

```
-- Get maximum TTL
cache.max_ttl()
518400
-- Set maximum TTL
cache.max_ttl(172800)
172800
```

cache.min_ttl([ttl])

Parameters

- **ttl** (*number*) – minimum cache TTL in seconds (default: 5 seconds)

Returns current maximum TTL

Get or set minimum cache TTL. Any entry inserted into cache with TTL lower than minimal will be overridden to minimum TTL. Forcing TTL higher than specified violates DNS standards, use with care.

Note: The *ttl* value must be in range *<0, max_ttl)*.

Warning: This settings applies only to currently open cache, it will not persist if the cache is closed or reopened.

```
-- Get minimum TTL
cache.min_ttl()
0
-- Set minimum TTL
cache.min_ttl(5)
5
```

cache.ns_tout ([timeout])

Parameters

- **timeout** (*number*) – NS retry interval in milliseconds (default: *KR_NS_TIMEOUT_RETRY_INTERVAL*)

Returns current timeout

Get or set time interval for which a nameserver address will be ignored after determining that it doesn't return (useful) answers. The intention is to avoid waiting if there's little hope; instead, kresd can immediately SERVFAIL or immediately use stale records (with *serve_stale* module).

Warning: This settings applies only to the current kresd process.

cache.get ([domain])

This function is not implemented at this moment. We plan to re-introduce it soon, probably with a slightly different API.

cache.clear ([name], [exact_name], [rr_type], [chunk_size], [callback], [prev_state])

Purge cache records matching specified criteria. There are two specifics:

- To reliably remove **negative** cache entries you need to clear subtree with the whole zone. E.g. to clear negative cache entries for (formerly non-existing) record *www.example.com. A* you need to flush whole subtree starting at zone apex, e.g. *example.com.*[1].
- This operation is asynchronous and might not be yet finished when call to `cache.clear()` function returns. Return value indicates if clearing continues asynchronously or not.

Parameters

[1] This is a consequence of DNSSEC negative cache which relies on proofs of non-existence on various owner nodes. It is impossible to efficiently flush part of DNS zones signed with NSEC3.

- **name** (*string*) – subtree to purge; if the name isn't provided, whole cache is purged (and any other parameters are disregarded).
- **exact_name** (*bool*) – if set to `true`, only records with *the same* name are removed; default: false.
- **rr_type** (*kres.type*) – you may additionally specify the type to remove, but that is only supported with `exact_name == true`; default: nil.
- **chunk_size** (*integer*) – the number of records to remove in one round; default: 100. The purpose is not to block the resolver for long. The default `callback` repeats the command after one millisecond until all matching data are cleared.
- **callback** (*function*) – a custom code to handle result of the underlying C call. Its parameters are copies of those passed to *cache.clear()* with one additional parameter `rettable` containing table with return value from current call. `count` field contains a return code from *kr_cache_remove_subtree()*.
- **prev_state** (*table*) – return value from previous run (can be used by callback)

Return type table

Returns

`count` key is always present. Other keys are optional and their presence indicate special conditions.

- **count** *(integer)* - number of items removed from cache by this call (can be 0 if no entry matched criteria)
- **not_apex** - cleared subtree is not cached as zone apex; proofs of non-existence were probably not removed
- **subtree** *(string)* - hint where zone apex lies (this is estimation from cache content and might not be accurate)
- **chunk_limit** - more than `chunk_size` items needs to be cleared, clearing will continue asynchronously

Examples:

```
-- Clear whole cache
> cache.clear()
[count] => 76

-- Clear records at and below 'com.'
> cache.clear('com.')
[chunk_limit] => chunk size limit reached; the default callback will continue␣
↪asynchronously
[not_apex] => to clear proofs of non-existence call cache.clear('com.')
[count] => 100
[round] => 1
[subtree] => com.
> worker.sleep(0.1)
[cache] asynchonous cache.clear('com', false) finished

-- Clear only 'www.example.com.'
> cache.clear('www.example.com.', true)
[round] => 1
[count] => 1
```

(continues on next page)

(continued from previous page)

```
[not_apex] => to clear proofs of non-existence call cache.clear('example.com.')
[subtree] => example.com.
```

6.2 Multiple instances

Note: This section describes the usage of kresd when running under systemd. For other uses, please refer to *Usage without systemd*.

Knot Resolver can utilize multiple CPUs running in multiple independent instances (processes), where each process utilizes at most single CPU core on your machine. If your machine handles a lot of DNS traffic run multiple instances.

All instances typically share the same configuration and cache, and incomming queries are automatically distributed by operating system among all instances.

Advantage of using multiple instances is that a problem in a single instance will not affect others, so a single instance crash will not bring whole DNS resolver service down.

Tip: For maximum performance, there should be as many kresd processes as there are available CPU threads.

To run multiple instances, use a different identifier after @ sign for each instance, for example:

```
$ systemctl start kresd@1.service
$ systemctl start kresd@2.service
$ systemctl start kresd@3.service
$ systemctl start kresd@4.service
```

With the use of brace expansion in BASH the equivalent command looks like this:

```
$ systemctl start kresd@{1..4}.service
```

For more details see `kresd.systemd(7)`.

6.2.1 Zero-downtime restarts

Resolver restart normally takes just miliseconds and cache content is persistent to avoid performance drop after restart. If you want real zero-downtime restarts use *multiple instances* and do rolling restart, i.e. restart only one resolver process at a time.

On a system with 4 instances run these commands sequentially:

```
$ systemctl restart kresd@1.service
$ systemctl restart kresd@2.service
$ systemctl restart kresd@3.service
$ systemctl restart kresd@4.service
```

At any given time only a single instance is stopped and restarted so remaining three instances continue to service clients.

6.2.2 Instance-specific configuration

Instances can use arbitrary identifiers for the instances, for example we can name instances like *dns1*, *tls* and so on.

```
$ systemctl start kresd@dns1
$ systemctl start kresd@dns2
$ systemctl start kresd@tls
$ systemctl start kresd@doh
```

The instance name is subsequently exposed to kresd via the environment variable `SYSTEMD_INSTANCE`. This can be used to tell the instances apart, e.g. when using the *Name Server Identifier (NSID)* module with per-instance configuration:

```
local systemd_instance = os.getenv("SYSTEMD_INSTANCE")

modules.load('nsid')
nsid.name(systemd_instance)
```

More arcane set-ups are also possible. The following example isolates the individual services for classic DNS, DoT and DoH from each other.

```
local systemd_instance = os.getenv("SYSTEMD_INSTANCE")

if string.match(systemd_instance, '^dns') then
     net.listen('127.0.0.1', 53, { kind = 'dns' })
elseif string.match(systemd_instance, '^tls') then
     net.listen('127.0.0.1', 853, { kind = 'tls' })
elseif string.match(systemd_instance, '^doh') then
     net.listen('127.0.0.1', 443, { kind = 'doh' })
else
     panic("Use kresd@dns*, kresd@tls* or kresd@doh* instance names")
end
```

6.3 Prefetching records

The module refreshes records that are about to expire when they're used (having less than 1% of original TTL). This improves latency for frequently used records, as they are fetched in advance.

It is also able to learn usage patterns and repetitive queries that the server makes. For example, if it makes a query every day at 18:00, the resolver expects that it is needed by that time and prefetches it ahead of time. This is helpful to minimize the perceived latency and keeps the cache hot.

Tip: The tracking window and period length determine memory requirements. If you have a server with relatively fast query turnover, keep the period low (hour for start) and shorter tracking window (5 minutes). For personal slower resolver, keep the tracking window longer (i.e. 30 minutes) and period longer (a day), as the habitual queries occur daily. Experiment to get the best results.

6.3.1 Example configuration

```
modules = {
        predict = {
```

(continues on next page)

(continued from previous page)

```
                window = 15, -- 15 minutes sampling window
                period = 6*(60/15) -- track last 6 hours
        }
}
```

Defaults are 15 minutes window, 6 hours period.

Tip: Use period 0 to turn off prediction and just do prefetching of expiring records. That works even without the *stats* module.

Note: Otherwise this module requires *stats* module and loads it if not present.

6.3.2 Exported metrics

To visualize the efficiency of the predictions, the module exports following statistics.

- `predict.epoch` - current prediction epoch (based on time of day and sampling window)
- `predict.queue` - number of queued queries in current window
- `predict.learned` - number of learned queries in current window

6.3.3 Properties

`predict.config`({ window = *15*, period = *24}*)
Reconfigure the predictor to given tracking window and period length. Both parameters are optional. Window length is in minutes, period is a number of windows that can be kept in memory. e.g. if a `window` is 15 minutes, a `period` of "24" means 6 hours.

6.4 Cache prefilling

This module provides ability to periodically prefill DNS cache by importing root zone data obtained over HTTPS.

Intended users of this module are big resolver operators which will benefit from decreased latencies and smaller amount of traffic towards DNS root servets.

Example configuration is:

```
modules.load('prefill')
prefill.config({
      ['.'] = {
              url = 'https://www.internic.net/domain/root.zone',
              interval = 86400  -- seconds
              ca_file = '/etc/pki/tls/certs/ca-bundle.crt', -- optional
      }
})
```

This configuration downloads zone file from URL *https://www.internic.net/domain/root.zone* and imports it into cache every 86400 seconds (1 day). The HTTPS connection is authenticated using CA certificate from file */etc/pki/tls/certs/ca-bundle.crt* and signed zone content is validated using DNSSEC.

Root zone to import must be signed using DNSSEC and the resolver must have valid DNSSEC configuration.

Parameter	Description
ca_file	path to CA certificate bundle used to authenticate the HTTPS connection (optional, system-wide store will be used if not specified)
interval	number of seconds between zone data refresh attempts
url	URL of a file in **RFC 1035** zone file format

Only root zone import is supported at the moment.

6.4.1 Dependencies

Depends on the prefill-lua-http and luafilesystem libraries.

6.5 Serve stale

Demo module that allows using timed-out records in case kresd is unable to contact upstream servers.

By default it allows stale-ness by up to one day, after roughly four seconds trying to contact the servers. It's quite configurable/flexible; see the beginning of the module source for details. See also the RFC draft (not fully followed) and *cache.ns_tout*.

6.5.1 Running

```
modules = { 'serve_stale < cache' }
```

6.6 Root on loopback (RFC 7706)

Knot Resolver developers think that literal implementation of **RFC 7706** ("Decreasing Access Time to Root Servers by Running One on Loopback") is a bad idea so it is not implemented in the form envisioned by the RFC.

You can get the very similar effect without its downsides by combining *Cache prefilling* and *Serve stale* modules with Aggressive Use of DNSSEC-Validated Cache (**RFC 8198**) behavior which is enabled automatically together with DNSSEC validation.

6.7 Priming module

The module for Initializing a DNS Resolver with Priming Queries implemented according to **RFC 8109**. Purpose of the module is to keep up-to-date list of root DNS servers and associated IP addresses.

Result of successful priming query replaces root hints distributed with the resolver software. Unlike other DNS resolvers, Knot Resolver caches result of priming query on disk and keeps the data between restarts until TTL expires.

This module is enabled by default and it is not recommended to disable it. For debugging purposes you may disable the module by appending `modules.unload('priming')` to your configuration.

6.8 EDNS keepalive

The `edns_keepalive` module implements **RFC 7828** for *clients* connecting to Knot Resolver via TCP and TLS. The module just allows clients to discover the connection timeout, client connections are always timed-out the same way *regardless* of clients sending the EDNS option.

When connecting to servers, Knot Resolver does not send this EDNS option. It still attempts to reuse established connections intelligently.

This module is loaded by default. For debugging purposes it can be unloaded using standard means:

```
modules.load('edns_keepalive')
```

CHAPTER 7

Policy, access control, data manipulation

Features in this section allow to configure what clients can get access to what DNS data, i.e. DNS data filtering and manipulation.

Query policies specify global policies applicable to all requests, e.g. for blocking access to particular domain. *Views and ACLs* allow to specify per-client policies, e.g. block or unblock access to a domain only for subset of clients.

It is also possible to modify data returned to clients, either by providing *Static hints* (answers with statically configured IP addresses), *DNS64* translation, or *IP address renumbering*.

Additional modules offer protection against various DNS-based attacks, see *Rebinding protection* and *Refuse queries without RD bit*.

At the very end, module *DNS Application Firewall* provides HTTP API for run-time policy modification, and generally just offers different interface for previously mentioned features.

7.1 Query policies

This module can block, rewrite, or alter inbound queries based on user-defined policies.

Each policy *rule* has two parts: a *filter* and an *action*. A *filter* selects which queries will be affected by the policy, and *action* which modifies queries matching the associated filter.

Typically a rule is defined as follows: `filter(action(action parameters), filter parameters)`. For example, a filter can be `suffix` which matches queries whose suffix part is in specified set, and one of possible actions is `DENY`, which denies resolution. These are combined together into `policy.suffix(policy.DENY, {todname('badguy.example.')})`. The rule is effective when it is added into rule table using `policy.add()`, please see *Policy examples*.

This module is enabled by default because it implements mandatory **RFC 6761** logic. When no rule applies to a query, built-in rules for special-use and locally-served domain names are applied. These rules can be overriden by action `PASS`, see *Policy examples* below. For debugging purposes you can also add `modules.unload('policy')` to your config to unload the module.

7.1.1 Filters

A *filter* selects which queries will be affected by specified *action*. There are several policy filters available in the `policy.` table:

- `all(action)` - always applies the action
- `pattern(action, pattern)` - applies the action if QNAME matches a regular expression
- `suffix(action, table)` - applies the action if QNAME suffix matches one of suffixes in the table (useful for "is domain in zone" rules), uses Aho-Corasick string matching algorithm from CloudFlare (BSD 3-clause)
- *`policy.suffix_common`*
- `rpz(default_action, path)` - implements a subset of RPZ in zonefile format. See below for details: *`policy.rpz`*.
- `slice(slice_func, action, action, ...)` - splits the entire domain space into multiple slices, uses the slicing function to determine to which slice does the query belong, and perfroms the corresponding action. For details, see *`policy.slice`*.
- custom filter function

7.1.2 Actions

An *action* is function which modifies DNS query, and is either of type *chain* or *non-chain*. So-called *chain* actions modify the query and allow other rules to evaluate and modify the same query. *Non-chain* actions have opposite behavior, i.e. modify the query and stop rule processing.

Resolver comes with several actions available in the `policy.` table:

Non-chain actions

Following actions stop the policy matching on the query, i.e. other rules are not evaluated once rule with following actions matches:

- `PASS` - let the query pass through; it's useful to make exceptions before wider rules
- `DENY` - reply NXDOMAIN authoritatively
- `DENY_MSG(msg)` - reply NXDOMAIN authoritatively and add explanatory message to additional section
- `DROP` - terminate query resolution and return SERVFAIL to the requestor
- `REFUSE` - terminate query resolution and return REFUSED to the requestor
- `TC` - set TC=1 if the request came through UDP, forcing client to retry with TCP
- `FORWARD(ip)` - resolve a query via forwarding to an IP while validating and caching locally
- `TLS_FORWARD({{ip, authentication}})` - resolve a query via TLS connection forwarding to an IP while validating and caching locally
- `STUB(ip)` - similar to `FORWARD(ip)` but *without* attempting DNSSEC validation. Each request may be either answered from cache or simply sent to one of the IPs with proxying back the answer.
- `REROUTE({{subnet,target}, ...})` - reroute addresses in response matching given subnet to given target, e.g. `{'192.0.2.0/24', '127.0.0.0'}` will rewrite '192.0.2.55' to '127.0.0.55', see *renumber module* for more information.

`FORWARD`, `TLS_FORWARD` and `STUB` support up to four IP addresses "in a single call".

Chain actions

Following actions allow to keep trying to match other rules, until a non-chain action is triggered:

- `MIRROR(ip)` - mirror query to given IP and continue solving it (useful for partial snooping).
- `QTRACE` - pretty-print DNS response packets into the log for the query and its sub-queries. It's useful for debugging weird DNS servers.
- `FLAGS(set, clear)` - set and/or clear some flags for the query. There can be multiple flags to set/clear. You can just pass a single flag name (string) or a set of names.

Also, it is possible to write your own action (i.e. Lua function). It is possible to implement complex heuristics, e.g. to deflect Slow drip DNS attacks or gray-list resolution of misbehaving zones.

Warning: The policy module currently only looks at whole DNS requests. The rules won't be re-applied e.g. when following CNAMEs.

Note: The module (and `kres`) expects domain names in wire format, not textual representation. So each label in name is prefixed with its length, e.g. "example.com" equals to `"\7example\3com"`. You can use convenience function `todname('example.com')` for automatic conversion.

7.1.3 Forwarding over TLS protocol (DNS-over-TLS)

Policy *TLS_FORWARD* allows you to forward queries using Transport Layer Security protocol, which hides the content of your queries from an attacker observing the network traffic. Further details about this protocol can be found in **RFC 7858** and IETF draft dprive-dtls-and-tls-profiles.

Queries affected by *TLS_FORWARD* policy will always be resolved over TLS connection. Knot Resolver does not implement fallback to non-TLS connection, so if TLS connection cannot be established or authenticated according to the configuration, the resolution will fail.

To test this feature you need to either *configure Knot Resolver as DNS-over-TLS server*, or pick some public DNS-over-TLS server. Please see DNS Privacy Project homepage for list of public servers.

Note: Some public DNS-over-TLS providers may apply rate-limiting which makes their service incompatible with Knot Resolver's TLS forwarding. Notably, Google Public DNS doesn't work as of 2019-07-10.

When multiple servers are specified, the one with the lowest round-trip time is used.

CA+hostname authentication

Traditional PKI authentication requires server to present certificate with specified hostname, which is issued by one of trusted CAs. Example policy is:

```
policy.TLS_FORWARD({
        {'2001:DB8::d0c', hostname='res.example.com'}})
```

- `hostname` must be a valid domain name matching server's certificate. It will also be sent to the server as SNI.

- `ca_file` optionally contains a path to a CA certificate (or certificate bundle) in PEM format. If you omit that, the system CA certificate store will be used instead (usually sufficient). A list of paths is also accepted, but all of them must be valid PEMs.

Key-pinned authentication

Instead of CAs, you can specify hashes of accepted certificates in `pin_sha256`. They are in the usual format – base64 from sha256. You may still specify `hostname` if you want SNI to be sent.

TLS Examples

```
modules = { 'policy' }
-- forward all queries over TLS to the specified server
policy.add(policy.all(policy.TLS_FORWARD({{'192.0.2.1', pin_sha256='YQ=='}})))
-- for brevity, other TLS examples omit policy.add(policy.all())
-- single server authenticated using its certificate pin_sha256
  policy.TLS_FORWARD({{'192.0.2.1', pin_sha256='YQ=='}})  -- pin_sha256 is base64-
↪encoded
-- single server authenticated using hostname and system-wide CA certificates
  policy.TLS_FORWARD({{'192.0.2.1', hostname='res.example.com'}})
-- single server using non-standard port
  policy.TLS_FORWARD({{'192.0.2.1@443', pin_sha256='YQ=='}})  -- use @ or # to␣
↪specify port
-- single server with multiple valid pins (e.g. anycast)
  policy.TLS_FORWARD({{'192.0.2.1', pin_sha256={'YQ==', 'Wg=='}})
-- multiple servers, each with own authenticator
  policy.TLS_FORWARD({ -- please note that { here starts list of servers
        {'192.0.2.1', pin_sha256='Wg=='},
        -- server must present certificate issued by specified CA and hostname must␣
↪match
        {'2001:DB8::d0c', hostname='res.example.com', ca_file='/etc/knot-resolver/
↪tlsca.crt'}
})
```

Forwarding to multiple targets

With the use of *`policy.slice`* function, it is possible to split the entire DNS namespace into distinct slices. When used in conjuction with `policy.TLS_FORWARD`, it's possible to forward different queries to different targets.

```
policy.add(policy.slice(
    policy.slice_randomize_psl(),
    policy.TLS_FORWARD({{'192.0.2.1', hostname='res.example.com'}}),
    policy.TLS_FORWARD({
        -- multiple servers can be specified for a single slice
        -- the one with lowest round-trip time will be used
        {'193.17.47.1', hostname='odvr.nic.cz'},
        {'185.43.135.1', hostname='odvr.nic.cz'},
    })
))
```

Note: The privacy implications of using this feature aren't clear. Since websites often make requests to multiple domains, these might be forwarded to different targets. This could result in decreased privacy (e.g. when the remote targets are both logging or otherwise processing your DNS traffic). The intended use-case is to use this feature with

semi-trusted resolvers which claim to do no logging (such as those listed on dnsprivacy.org), to decrease the potential exposure of your DNS data to a malicious resolver operator.

7.1.4 Policy examples

```
-- Whitelist 'www[0-9].badboy.cz'
policy.add(policy.pattern(policy.PASS, '\4www[0-9]\6badboy\2cz'))
-- Block all names below badboy.cz
policy.add(policy.suffix(policy.DENY, {todname('badboy.cz.')}))

-- Custom rule
local ffi = require('ffi')
local function genRR (state, req)
        local answer = req.answer
        local qry = req:current()
        if qry.stype ~= kres.type.A then
                return state
        end
        ffi.C.kr_pkt_make_auth_header(answer)
        answer:rcode(kres.rcode.NOERROR)
        answer:begin(kres.section.ANSWER)
        answer:put(qry.sname, 900, answer:qclass(), kres.type.A, '\192\168\1\3')
        return kres.DONE
end
policy.add(policy.suffix(genRR, { todname('my.example.cz.') }))

-- Disallow ANY queries
policy.add(function (req, query)
        if query.stype == kres.type.ANY then
                return policy.DROP
        end
end)
-- Enforce local RPZ
policy.add(policy.rpz(policy.DENY, 'blacklist.rpz'))
-- Forward all queries below 'company.se' to given resolver;
-- beware: typically this won't work due to DNSSEC - see "Replacing part..." below
policy.add(policy.suffix(policy.FORWARD('192.168.1.1'), {todname('company.se')}))
-- Forward reverse queries about the 192.168.1.1/24 space to .1 port 5353
-- and do it directly without attempts to validate DNSSEC etc.
policy.add(policy.suffix(policy.STUB('192.168.1.1@5353'), {todname('1.168.192.in-addr.
↪arpa')}))
-- Forward all queries matching pattern
policy.add(policy.pattern(policy.FORWARD('2001:DB8::1'), '\4bad[0-9]\2cz'))
-- Forward all queries (to public resolvers https://www.nic.cz/odvr)
policy.add(policy.all(policy.FORWARD({'2001:148f:fffe::1', '193.14.47.1'})))
-- Print all responses with matching suffix
policy.add(policy.suffix(policy.QTRACE, {todname('rhybar.cz.')}))
-- Print all responses
policy.add(policy.all(policy.QTRACE))
-- Mirror all queries and retrieve information
local rule = policy.add(policy.all(policy.MIRROR('127.0.0.2')))
-- Print information about the rule
print(string.format('id: %d, matched queries: %d', rule.id, rule.count)
-- Reroute all addresses found in answer from 192.0.2.0/24 to 127.0.0.x
-- this policy is enforced on answers, therefore 'postrule'
```

(continues on next page)

(continued from previous page)

```
local rule = policy.add(policy.REROUTE({'192.0.2.0/24', '127.0.0.0'}), true)
-- Delete rule that we just created
policy.del(rule.id)
```

7.1.5 Replacing part of the DNS tree

You may want to resolve most of the DNS namespace by usual means while letting some other resolver solve specific subtrees. Such data would typically be rejected by DNSSEC validation starting from the ICANN root keys. Therefore, if you trust the resolver and your link to it, you can simply use the `STUB` action instead of `FORWARD` to avoid validation only for those subtrees.

Another issue is caused by caching, because Knot Resolver only keeps a single cache for everything. For example, if you add an alternative top-level domain while using the ICANN root zone for the rest, at some point the cache may obtain records proving that your top-level domain does not exist, and those records could then be used when the positive records fall out of cache. The easiest work-around is to disable reading from cache for those subtrees; the other resolver is often very close anyway.

Listing 1: Example configuration: graft DNS sub-trees `faketldtest`, `sld.example`, and `internal.example.com` into existing namespace

```
extraTrees = policy.todnames({'faketldtest', 'sld.example', 'internal.example.com'})
-- Beware: the rule order is important, as STUB is not a chain action.
policy.add(policy.suffix(policy.FLAGS({'NO_CACHE'}),   extraTrees))
policy.add(policy.suffix(policy.STUB({'2001:db8::1'}), extraTrees))
```

7.1.6 Additional properties

Most properties (actions, filters) are described above.

policy.add(rule, postrule)

Parameters

- **rule** – added rule, i.e. `policy.pattern(policy.DENY, '[0-9]+\2cz')`
- **postrule** – boolean, if true the rule will be evaluated on answer instead of query

Returns rule description

Add a new policy rule that is executed either or queries or answers, depending on the `postrule` parameter. You can then use the returned rule description to get information and unique identifier for the rule, as well as match count.

policy.del(id)

Parameters

- **id** – identifier of a given rule

Returns boolean

Remove a rule from policy list.

policy.suffix_common(action, suffix_table[, common_suffix])

Parameters

- **action** – action if the pattern matches QNAME
- **suffix_table** – table of valid suffixes
- **common_suffix** – common suffix of entries in suffix_table

Like suffix match, but you can also provide a common suffix of all matches for faster processing (nil otherwise). This function is faster for small suffix tables (in the order of "hundreds").

policy.rpz (action, path, watch)

Parameters

- **action** – the default action for match in the zone; typically you want `policy.DENY`
- **path** – path to zone file | database
- **watch** – boolean, if not false, the file will be reparsed and the ruleset reloaded on file change

Enforce RPZ rules. This can be used in conjunction with published blocklist feeds. The RPZ operation is well described in this Jan-Piet Mens's post, or the Pro DNS and BIND book. Here's compatibility table:

Policy Action	RH Value	Support
`action` is used	`.`	**yes**, if `action` is `DENY`
`action` is used	`*.`	*partial*[1]
`policy.PASS`	`rpz-passthru.`	**yes**
`policy.DROP`	`rpz-drop.`	**yes**
`policy.TC`	`rpz-tcp-only.`	**yes**
Modified	anything	no

Policy Trigger	Support
QNAME	**yes**
CLIENT-IP	*partial*, may be done with *views*
IP	no
NSDNAME	no
NS-IP	no

policy.slice (slice_func, action[, action[, ...])

Parameters

- **slice_func** – slicing function that returns index based on query
- **action** – action to be performed for the slice

This function splits the entire domain space into multiple slices (determined by the number of provided `actions`). A `slice_func` is called to determine which slice a query belongs to. The corresponding `action` is then executed.

policy.slice_randomize_psl (seed = os.time() / (3600 * 24 * *7*))

Parameters

- **seed** – seed for random assignment

The function initializes and returns a slicing function, which deterministically assigns `query` to a slice based on the QNAME.

[1] The specification for `*.` wants a `NODATA` answer. For now, `policy.DENY` action doing `NXDOMAIN` is typically used instead.

It utilizes the Public Suffix List to ensure domains under the same registrable domain end up in a single slice. (see example below)

`seed` can be used to re-shuffle the slicing algorhitm when the slicing function is initialized. By default, the assigment is re-shuffled after one week (when resolver restart / reloads config). To force a stable distribution, pass a fixed value. To re-shuffle on every resolver restart, use `os.time()`.

The following example demonstrates a distribution among 3 slices:

```
slice 1/3:
example.com
a.example.com
b.example.com
x.b.example.com
example3.com

slice 2/3:
example2.co.uk

slice 3/3:
example.co.uk
a.example.co.uk
```

policy.todnames({name, ...})

Param names table of domain names in textual format

Returns table of domain names in wire format converted from strings.

```
-- Convert single name
assert(todname('example.com') == '\7example\3com\0')
-- Convert table of names
policy.todnames({'example.com', 'me.cz'})
{ '\7example\3com\0', '\2me\2cz\0' }
```

7.2 Views and ACLs

The *policy* module implements policies for global query matching, e.g. solves "how to react to certain query". This module combines it with query source matching, e.g. "who asked the query". This allows you to create personalized blacklists, filters and ACLs.

There are two identification mechanisms:

- `addr` - identifies the client based on his subnet
- `tsig` - identifies the client based on a TSIG key name (only for testing purposes, TSIG signature is not verified!)

View module allows you to combine query source information with *policy* rules.

```
view:addr('10.0.0.1', policy.suffix(policy.TC, policy.todnames({'example.com'})))
```

This example will force given client to TCP for names in `example.com` subtree. You can combine view selectors with RPZ to create personalized filters for example.

Warning: Beware that cache is shared by *all* requests. For example, it is safe to refuse answer based on who asks the resolver, but trying to serve different data to different clients will result in unexpected behavior. Setups like **split-horizon** which depend on isolated DNS caches are explicitly not supported.

7.2.1 Example configuration

```
-- Load modules
modules = { 'view' }
-- Whitelist queries identified by TSIG key
view:tsig('\5mykey', policy.all(policy.PASS))
-- Block local IPv4 clients (ACL like)
view:addr('127.0.0.1', policy.all(policy.DENY))
-- Block local IPv6 clients (ACL like)
view:addr('::1', policy.all(policy.DENY))
-- Drop queries with suffix match for remote client
view:addr('10.0.0.0/8', policy.suffix(policy.DROP, policy.todnames({'xxx'})))
-- RPZ for subset of clients
view:addr('192.168.1.0/24', policy.rpz(policy.PASS, 'whitelist.rpz'))
-- Do not try this - it will pollute cache and surprise you!
-- view:addr('10.0.0.0/8', policy.all(policy.FORWARD('2001:DB8::1')))
-- Drop everything that hasn't matched
view:addr('0.0.0.0/0', policy.all(policy.DROP))
```

7.2.2 Rule order

The current implementation is best understood as three separate rule chains: vanilla `policy.add`, `view:tsig` and `view:addr`. For each request the rules in these chains get tried one by one until a *non-chain policy action* gets executed.

By default *policy module* acts before `view` module due to `policy` being loaded by default. If you want to intermingle universal rules with `view:addr`, you may simply wrap the universal policy rules in view closure like this:

```
view:addr('0.0.0.0/0', policy.<rule>) -- and
view:addr('::0/0',     policy.<rule>)
```

7.2.3 Properties

view:addr(subnet, rule)

Parameters

- **subnet** – client subnet, i.e. `10.0.0.1`
- **rule** – added rule, i.e. `policy.pattern(policy.DENY, '[0-9]+\2cz')`

Apply rule to clients in given subnet.

view:tsig(key, rule)

Parameters

- **key** – client TSIG key domain name, i.e. `\5mykey`
- **rule** – added rule, i.e. `policy.pattern(policy.DENY, '[0-9]+\2cz')`

Apply rule to clients with given TSIG key.

> **Warning:** This just selects rule based on the key name, it doesn't verify the key or signature yet.

7.3 Static hints

This is a module providing static hints for forward records (A/AAAA) and reverse records (PTR). The records can be loaded from `/etc/hosts`-like files and/or added directly.

You can also use the module to change the root hints; they are used as a safety belt or if the root NS drops out of cache.

7.3.1 Examples

```
-- Load hints after iterator (so hints take precedence before caches)
modules = { 'hints > iterate' }
-- Add a custom hosts file
hints.add_hosts('hosts.custom')
-- Override the root hints
hints.root({
  ['j.root-servers.net.'] = { '2001:503:c27::2:30', '192.58.128.30' }
})
-- Add a custom hint
hints['foo.bar'] = '127.0.0.1'
```

Note: The *policy* module applies before hints, meaning e.g. that hints for special names (**RFC 6761#section-6**) like `localhost` or `test` will get shadowed by policy rules by default. That can be worked around e.g. by explicit `policy.PASS` action.

7.3.2 Properties

`hints.config`([path])

Parameters

- **`path`** (*`string`*) – path to hosts-like file, default: no file

Returns `{ result:  bool }`

Clear any configured hints, and optionally load a hosts-like file as in `hints.add_hosts(path)`. (Root hints are not touched.)

`hints.add_hosts`([path])

Parameters

- **`path`** (*`string`*) – path to hosts-like file, default: `/etc/hosts`

Add hints from a host-like file.

`hints.get`(*hostname*)

Parameters

- **`hostname`** (*`string`*) – i.e. `"localhost"`

Returns `{ result:  [address1, address2, ...] }`

Return list of address record matching given name. If no hostname is specified, all hints are returned in the table format used by `hints.root()`.

`hints.set`(pair)

Parameters

- **pair** (*string*) – `hostname address` i.e. `"localhost 127.0.0.1"`

Returns `{ result: bool }`

Add a hostname–address pair hint.

Note: If multiple addresses have been added for a name (in separate `hints.set()` commands), all are returned in a forward query. If multiple names have been added to an address, the last one defined is returned in a corresponding PTR query.

---

hints.del(pair)

Parameters

- **pair** (*string*) – `hostname address` i.e. `"localhost 127.0.0.1"`, or just `hostname`

Returns `{ result: bool }`

Remove a hostname - address pair hint. If address is omitted, all addresses for the given name are deleted.

hints.root_file(path)

Replace current root hints from a zonefile. If the path is omitted, the compiled-in path is used, i.e. the root hints are reset to the default.

hints.root(root_hints)

Parameters

- **root_hints** (*table*) – new set of root hints i.e. `{['name'] = 'addr', ...}`

Returns `{ ['a.root-servers.net.'] = { '1.2.3.4', '5.6.7.8', ...}, ... }`

Replace current root hints and return the current table of root hints.

Tip: If no parameters are passed, it only returns current root hints set without changing anything.

Example:

```
> hints.root({
  ['l.root-servers.net.'] = '199.7.83.42',
  ['m.root-servers.net.'] = '202.12.27.33'
})
[l.root-servers.net.] => {
  [1] => 199.7.83.42
}
[m.root-servers.net.] => {
  [1] => 202.12.27.33
}
```

Tip: A good rule of thumb is to select only a few fastest root hints. The server learns RTT and NS quality over time, and thus tries all servers available. You can help it by preselecting the candidates.

---

hints.use_nodata(toggle)

Parameters

- **toggle** (*bool*) – true if enabling NODATA synthesis, false if disabling

Returns `{ result: bool }`

If set to true (the default), NODATA will be synthesised for matching hint name, but mismatching type (e.g. AAAA query when only A hint exists).

hints.ttl ([new_ttl])

Parameters

- **new_ttl** (*int*) – new TTL to set (optional)

Returns the TTL setting

This function allows to read and write the TTL value used for records generated by the hints module.

7.4 DNS64

The module for **RFC 6147** DNS64 AAAA-from-A record synthesis, it is used to enable client-server communication between an IPv6-only client and an IPv4-only server. See the well written introduction in the PowerDNS documentation. If no address is passed (i.e. `nil`), the well-known prefix `64:ff9b::` is used.

Warning: The module currently won't work well with *policy.STUB*. Also, the IPv6 passed in configuration is assumed to be `/96`, and PTR synthesis and "exclusion prefixes" aren't implemented.

Tip: The A record sub-requests will be DNSSEC secured, but the synthetic AAAA records can't be. Make sure the last mile between stub and resolver is secure to avoid spoofing.

7.4.1 Example configuration

```
-- Load the module with a NAT64 address
modules = { dns64 = 'fe80::21b:77ff:0:0' }
-- Reconfigure later
dns64.config('fe80::21b:aabb:0:0')
```

7.5 IP address renumbering

The module renumbers addresses in answers to different address space. e.g. you can redirect malicious addresses to a blackhole, or use private address ranges in local zones, that will be remapped to real addresses by the resolver.

Warning: While requests are still validated using DNSSEC, the signatures are stripped from final answer. The reason is that the address synthesis breaks signatures. You can see whether an answer was valid or not based on the AD flag.

7.5.1 Example configuration

```
modules = {
        renumber = {
                -- Source subnet, destination subnet
                {'10.10.10.0/24', '192.168.1.0'},
                -- Remap /16 block to localhost address range
                {'166.66.0.0/16', '127.0.0.0'}
        }
}
```

7.6 Answer reordering

Certain clients are "dumb" and always connect to first IP address or name found in a DNS answer received from resolver intead of picking randomly. As a workaround for such broken clients it is possible to randomize order of records in DNS answers sent by resolver:

reorder_RR ([true | *false*])

Parameters

- **new_value** (*boolean*) – `true` to enable or `false` to disable randomization *(optional)*

Returns The (new) value of the option

If set, resolver will vary the order of resource records within RR sets. It is disabled by default.

7.7 Rebinding protection

This module provides protection from DNS Rebinding attack by blocking answers which cointain IPv4 or IPv6 addresses for private use (or some other special-use addresses).

To enable this module insert following line into your configuration file:

```
modules.load('rebinding < iterate')
```

Please note that this module does not offer stable configuration interface yet. For this reason it is suitable mainly for public resolver operators who do not need to whitelist certain subnets.

Warning: DNS Blacklists (RFC 5782) often use *127.0.0.0/8* to blacklist a domain. Using the rebinding module prevents DNSBL from functioning properly.

7.8 Refuse queries without RD bit

This module ensures all queries without RD (recursion desired) bit set in query are answered with REFUSED. This prevents snooping on the resolver's cache content.

The module is loaded by default. If you'd like to disable this behavior, you can unload it:

```
modules.unload('refuse_nord')
```

7.9 DNS Application Firewall

This module is a high-level interface for other powerful filtering modules and DNS views. It provides an easy interface to apply and monitor DNS filtering rules and a persistent memory for them. It also provides a restful service interface and an HTTP interface.

7.9.1 Example configuration

Firewall rules are declarative and consist of filters and actions. Filters have `field operator operand` notation (e.g. `qname = example.com`), and may be chained using AND/OR keywords. Actions may or may not have parameters after the action name.

```
-- Let's write some daft rules!
modules = { 'daf' }

-- Block all queries with QNAME = example.com
daf.add 'qname = example.com deny'

-- Filters can be combined using AND/OR...
-- Block all queries with QNAME match regex and coming from given subnet
daf.add 'qname ~ %w+.example.com AND src = 192.0.2.0/24 deny'

-- We also can reroute addresses in response to alternate target
-- This reroutes 1.2.3.4 to localhost
daf.add 'src = 127.0.0.0/8 reroute 192.0.2.1-127.0.0.1'

-- Subnets work too, this reroutes a whole subnet
-- e.g. 192.0.2.55 to 127.0.0.55
daf.add 'src = 127.0.0.0/8 reroute 192.0.2.0/24-127.0.0.0'

-- This rewrites all A answers for 'example.com' from
-- whatever the original address was to 127.0.0.2
daf.add 'src = 127.0.0.0/8 rewrite example.com A 127.0.0.2'

-- Mirror queries matching given name to DNS logger
daf.add 'qname ~ %w+.example.com mirror 127.0.0.2'
daf.add 'qname ~ example-%d.com mirror 127.0.0.3@5353'

-- Forward queries from subnet
daf.add 'src = 127.0.0.1/8 forward 127.0.0.1@5353'
-- Forward to multiple targets
daf.add 'src = 127.0.0.1/8 forward 127.0.0.1@5353,127.0.0.2@5353'

-- Truncate queries based on destination IPs
daf.add 'dst = 192.0.2.51 truncate'

-- Disable a rule
daf.disable 2
-- Enable a rule
daf.enable 2
-- Delete a rule
daf.del 2
```

If you're not sure what firewall rules are in effect, see `daf.rules`:

```
-- Show active rules
> daf.rules
[1] => {
    [rule] => {
        [count] => 42
        [id] => 1
        [cb] => function: 0x1a3eda38
    }
    [info] => qname = example.com AND src = 127.0.0.1/8 deny
    [policy] => function: 0x1a3eda38
}
[2] => {
    [rule] => {
        [suspended] => true
        [count] => 123522
        [id] => 2
        [cb] => function: 0x1a3ede88
    }
    [info] => qname ~ %w+.facebook.com AND src = 127.0.0.1/8 deny...
    [policy] => function: 0x1a3ede88
}
```

7.9.2 Web interface

If you have *HTTP/2* loaded, the firewall automatically loads as a snippet. You can create, track, suspend and remove firewall rules from the web interface. If you load both modules, you have to load *daf* after *http*.

7.9.3 RESTful interface

The module also exports a RESTful API for operations over rule chains.

URL	HTTP Verb	Action
/daf	GET	Return JSON list of active rules.
/daf	POST	Insert new rule, rule string is expected in body. Returns rule information in JSON.
/daf/<id>	GET	Retrieve a rule matching given ID.
/daf/<id>	DELETE	Delete a rule matching given ID.
/daf/<id>/<prop>/<val>	PATCH	Modify given rule, for example /daf/3/active/false suspends rule 3.

This interface is used by the web interface for all operations, but you can also use it directly for testing.

```
# Get current rule set
$ curl -s -X GET http://localhost:8453/daf | jq .
{}

# Create new rule
$ curl -s -X POST -d "src = 127.0.0.1 pass" http://localhost:8453/daf | jq .
{
  "count": 0,
  "active": true,
  "info": "src = 127.0.0.1 pass",
  "id": 1
```

(continues on next page)

(continued from previous page)

```
}

# Disable rule
$ curl -s -X PATCH http://localhost:8453/daf/1/active/false | jq .
true

# Retrieve a rule information
$ curl -s -X GET http://localhost:8453/daf/1 | jq .
{
  "count": 4,
  "active": true,
  "info": "src = 127.0.0.1 pass",
  "id": 1
}

# Delete a rule
$ curl -s -X DELETE http://localhost:8453/daf/1 | jq .
true
```

CHAPTER 8

Logging, monitoring, diagnostics

Knot Resolver logs to standard outputs, which is then captured by supervisor and sent to logging system for further processing. To read logs use commands usual for your distribution. E.g. on distributions using systemd-journald use command `journalctl -u kresd@* -f`.

During normal operation only errors and other very important events are logged, so by default logs from Knot Resolver should contain only couple lines a day. For debugging purposes it is possible to enable very verbose logging using *`verbose()`* function.

verbose ([true | *false*])

Param `true` to enable, `false` to disable verbose logging.

Returns boolean Current state of verbose logging.

Toggle global verbose logging. Use only for debugging purposes. On busy systems vebose logging can produce several MB of logs per second and will slow down operation.

It is also possible to obtain verbose logs for *a single request*, see chapter *Debugging a single request*.

Less verbose logging for DNSSEC validation errors can be enabled using *DNSSEC validation failure logging* module.

Various statistics for monitoring purposes are available in *Statistics collector* module, including export to central systems like Graphite, Metronome, InfluxDB, or Prometheus format.

Resolver *Watchdog* is tool to detect and recover from potential bugs that cause the resolver to stop responding properly to queries.

Additional monitoring and debugging methods are described below. If none of these options fits your deployment or if you have special needs you can configure your own checks and exports using *Asynchronous events*.

8.1 DNSSEC validation failure logging

This module adds error message for each DNSSEC validation failure. It is meant to provide hint to operators which queries should be investigated using diagnostic tools like DNSViz.

Add following line to your configuration file to enable it:

```
modules.load('bogus_log')
```

Example of error message logged by this module:

```
DNSSEC validation failure dnssec-failed.org. DNSKEY
```

List of most frequent queries which fail as DNSSEC bogus can be obtained at run-time:

```
> bogus_log.frequent()
[1] => {
    [type] => DNSKEY
    [count] => 1
    [name] => dnssec-failed.org.
}
[2] => {
    [type] => DNSKEY
    [count] => 13
    [name] => rhybar.cz.
}
```

Please note that in future this module might be replaced with some other way to log this information.

8.2 Statistics collector

Module `stats` gathers various counters from the query resolution and server internals, and offers them as a key-value storage. These metrics can be either exported to *Graphite/InfluxDB/Metronome*, exposed as *Prometheus metrics endpoint*, or processed using user-provided script as described in chapter *Asynchronous events*.

Note: Please remember that each Knot Resolver instance keeps its own statistics, and instances can be started and stopped dynamically. This might affect your data postprocessing procedures if you are using *Multiple instances*.

8.2.1 Built-in statistics

Built-in counters keep track of number of queries and answers matching specific criteria.

Global request counters	
request.total	total number of DNS requests (including internal client requests)
request.internal	internal requests generated by Knot Resolver (e.g. DNSSEC trust anchor updates)
request.udp	external requests received over plain UDP (**RFC 1035**)
request.tcp	external requests received over plain TCP (**RFC 1035**)
request.dot	external requests received over DNS-over-TLS (**RFC 7858**)
request.doh	external requests received over DNS-over-HTTP (**RFC 8484**)

Global answer counters	
answer.total	total number of answered queries
answer.cached	queries answered from cache

Answers categorized by RCODE	
answer.noerror	NOERROR answers
answer.nodata	NOERROR, but empty answers
answer.nxdomain	NXDOMAIN answers
answer.servfail	SERVFAIL answers

Answer latency	
answer.1ms	completed in 1ms
answer.10ms	completed in 10ms
answer.50ms	completed in 50ms
answer.100ms	completed in 100ms
answer.250ms	completed in 250ms
answer.500ms	completed in 500ms
answer.1000ms	completed in 1000ms
answer.1500ms	completed in 1500ms
answer.slow	completed in more than 1500ms

Answer flags	
answer.aa	authoritative answer
answer.tc	truncated answer
answer.ra	recursion available
answer.rd	recursion desired (in answer!)
answer.ad	authentic data (DNSSEC)
answer.cd	checking disabled (DNSSEC)
answer.do	DNSSEC answer OK
answer.edns0	EDNS0 present

Query flags	
query.edns	queries with EDNS present
query.dnssec	queries with DNSSEC DO=1

Example:

```
modules.load('stats')

-- Enumerate metrics
> stats.list()
[answer.cached] => 486178
[iterator.tcp] => 490
[answer.noerror] => 507367
[answer.total] => 618631
[iterator.udp] => 102408
[query.concurrent] => 149

-- Query metrics by prefix
> stats.list('iter')
[iterator.udp] => 105104
[iterator.tcp] => 490

-- Fetch most common queries
```

(continues on next page)

(continued from previous page)

```
> stats.frequent()
[1] => {
        [type] => 2
        [count] => 4
        [name] => cz.
}

-- Fetch most common queries (sorted by frequency)
> table.sort(stats.frequent(), function (a, b) return a.count > b.count end)

-- Show recently contacted authoritative servers
> stats.upstreams()
[2a01:618:404::1] => {
    [1] => 26 -- RTT
}
[128.241.220.33] => {
    [1] => 31 - RTT
}

-- Set custom metrics from modules
> stats['filter.match'] = 5
> stats['filter.match']
5
```

8.2.2 Module reference

stats.get(key)

Parameters

- **key** (*string*) – i.e. `"answer.total"`

Returns `number`

Return nominal value of given metric.

stats.set(key, val)

Parameters

- **key** (*string*) – i.e. `"answer.total"`
- **val** (*number*) – i.e. `5`

Set nominal value of given metric.

stats.list([prefix])

Parameters

- **prefix** (*string*) – optional metric prefix, i.e. `"answer"` shows only metrics beginning with "answer"

Outputs collected metrics as a JSON dictionary.

stats.upstreams()

Outputs a list of recent upstreams and their RTT. It is sorted by time and stored in a ring buffer of a fixed size. This means it's not aggregated and readable by multiple consumers, but also that you may lose entries if you don't read quickly enough. The default ring size is 512 entries, and may be overriden on compile time by `-DUPSTREAMS_COUNT=X`.

stats.frequent()

Outputs list of most frequent iterative queries as a JSON array. The queries are sampled probabilistically, and include subrequests. The list maximum size is 5000 entries, make diffs if you want to track it over time.

stats.clear_frequent()

Clear the list of most frequent iterative queries.

8.2.3 Graphite/InfluxDB/Metronome

The `graphite` sends statistics over the Graphite protocol to either Graphite, Metronome, InfluxDB or any compatible storage. This allows powerful visualization over metrics collected by Knot Resolver.

Tip: The Graphite server is challenging to get up and running, InfluxDB combined with Grafana are much easier, and provide richer set of options and available front-ends. Metronome by PowerDNS alternatively provides a mini-graphite server for much simpler setups.

Example configuration:

Only the `host` parameter is mandatory.

By default the module uses UDP so it doesn't guarantee the delivery, set `tcp = true` to enable Graphite over TCP. If the TCP consumer goes down or the connection with Graphite is lost, resolver will periodically attempt to reconnect with it.

```
modules = {
        graphite = {
                prefix = hostname(), -- optional metric prefix
                host = '127.0.0.1',  -- graphite server address
                port = 2003,         -- graphite server port
                interval = 5 * sec,  -- publish interval
                tcp = false          -- set to true if want TCP mode
        }
}
```

The module supports sending data to multiple servers at once.

```
modules = {
        graphite = {
                host = { '127.0.0.1', '1.2.3.4', '::1' },
        }
}
```

Dependencies

- lua cqueues package.

8.2.4 Prometheus metrics endpoint

The HTTP module exposes `/metrics` endpoint that serves metrics from *Statistics collector* in Prometheus text format. You can use it as soon as HTTP module is configured:

```
$ curl -k https://localhost:8453/metrics | tail
# TYPE latency histogram
latency_bucket{le=10} 2.000000
latency_bucket{le=50} 2.000000
latency_bucket{le=100} 2.000000
latency_bucket{le=250} 2.000000
latency_bucket{le=500} 2.000000
latency_bucket{le=1000} 2.000000
latency_bucket{le=1500} 2.000000
latency_bucket{le=+Inf} 2.000000
latency_count 2.000000
latency_sum 11.000000
```

You can namespace the metrics in configuration, using *http.prometheus.namespace* attribute:

```
modules.load('http')
-- Set Prometheus namespace
http.prometheus.namespace = 'resolver_'
```

You can also add custom metrics or rewrite existing metrics before they are returned to Prometheus client.

```
modules.load('http')
-- Add an arbitrary metric to Prometheus
http.prometheus.finalize = function (metrics)
        table.insert(metrics, 'build_info{version="1.2.3"} 1')
end
```

8.3 Name Server Identifier (NSID)

Module `nsid` provides server-side support for **RFC 5001** which allows DNS clients to request resolver to send back its NSID along with the reply to a DNS request. This is useful for debugging larger resolver farms (e.g. when using *Multiple instances*, anycast or load balancers).

NSID value can be configured in the resolver's configuration file:

```
modules.load('nsid')
nsid.name('instance 1')
```

Tip: When dealing with Knot Resolver running in *multiple instances* managed with systemd see *Instance-specific configuration*.

You can also obtain configured NSID value:

```
> nsid.name()
'instance 1'
```

The module can be disabled at run-time:

```
modules.unload('nsid')
```

8.4 Debugging a single request

The HTTP module provides `/trace` endpoint which allows to trace various aspects of the request execution. The basic mode allows you to resolve a query and trace verbose logs for it (and messages received):

```
$ curl https://localhost:8453/trace/e.root-servers.net
[ 8138] [iter] 'e.root-servers.net.' type 'A' created outbound query, parent id 0
[ 8138] [ rc ] => rank: 020, lowest 020, e.root-servers.net. A
[ 8138] [ rc ] => satisfied from cache
[ 8138] [iter] <= answer received:
;; ->>HEADER<<- opcode: QUERY; status: NOERROR; id: 8138
;; Flags: qr aa  QUERY: 1; ANSWER: 0; AUTHORITY: 0; ADDITIONAL: 0

;; QUESTION SECTION
e.root-servers.net.          A

;; ANSWER SECTION
e.root-servers.net.  3556353 A       192.203.230.10

[ 8138] [iter] <= rcode: NOERROR
[ 8138] [resl] finished: 4, queries: 1, mempool: 81952 B
```

See chapter about HTTP module for further instructions how to load `webmgmt` endpoint into HTTP module, it is a prerequisite for using `/trace`.

8.5 Watchdog

This module cooperates with Systemd watchdog to restart the process in case the internal event loop gets stuck. The upstream Systemd unit files are configured to use this feature, which is turned on with the `WatchdogSec=` directive in the service file.

As an optional feature, this module can also do an internal DNS query to check if resolver answers correctly. To use this feature you must configure DNS name and type to query for:

```
watchdog.config({ qname = 'nic.cz.', qtype = kres.type.A })
```

Each single query from watchdog must result in answer with RCODE = NOERROR or NXDOMAIN. Any other result will terminate the resolver (with SIGABRT) to allow the supervisor process to do cleanup, gather coredump and restart the resolver.

It is recommended to use a name with a very short TTL to make sure the watchdog is testing all parts of resolver and not only its cache. Obviously this check makes sense only when used with very reliable domains; otherwise a failure on authoritative side will shutdown resolver!

WatchdogSec specifies deadline for supervisor when the process will be killed. Watchdog queries are executed each *WatchdogSec / 2* seconds. This implies that **half** of *WatchdogSec* interval must be long enough for normal DNS query to succeed, so do not forget to add two or three seconds for random network timeouts etc.

The module is loaded by default. If you'd like to disable it you can unload it:

```
modules.unload('watchdog')
```

Beware that unloading the module without disabling watchdog feature in supervisor will lead to infinite restart loop.

8.6 Dnstap (traffic collection)

The `dnstap` module supports logging DNS responses to a unix socket in dnstap format using fstrm framing library. This logging is useful if you need effectivelly log all DNS traffic.

The unix socket and the socket reader must be present before starting resolver instances.

Tunables:

- `socket_path`: the the unix socket file where dnstap messages will be sent
- `log_responses`: if `true` responses in wire format will be logged

```
modules = {
    dnstap = {
        socket_path = "/tmp/dnstap.sock",
        log_responses = true
    }
}
```

8.7 Sentinel for Detecting Trusted Root Keys

The module `ta_sentinel` implements A Root Key Trust Anchor Sentinel for DNSSEC according to standard **RFC 8509**.

This feature allows users of DNSSEC validating resolver to detect which root keys are configured in resolver's chain of trust. The data from such signaling are necessary to monitor the progress of the DNSSEC root key rollover and to detect potential breakage before it affect users. One example of research enabled by this module is available here.

This module is enabled by default and we urge users not to disable it. If it is absolutely necessary you may add `modules.unload('ta_sentinel')` to your configuration to disable it.

8.8 Signaling Trust Anchor Knowledge in DNSSEC

The module for Signaling Trust Anchor Knowledge in DNSSEC Using Key Tag Query, implemented according to **RFC 8145#section-5**.

This feature allows validating resolvers to signal to authoritative servers which keys are referenced in their chain of trust. The data from such signaling allow zone administrators to monitor the progress of rollovers in a DNSSEC-signed zone.

This mechanism serve to measure the acceptance and use of new DNSSEC trust anchors and key signing keys (KSKs). This signaling data can be used by zone administrators as a gauge to measure the successful deployment of new keys. This is of particular interest for the DNS root zone in the event of key and/or algorithm rollovers that rely on **RFC 5011** to automatically update a validating DNS resolver's trust anchor.

> **Attention:** Experience from root zone KSK rollover in 2018 shows that this mechanism by itself is not sufficient to reliably measure acceptance of the new key. Nevertheless, some DNS researchers found it is useful in combination with other data so we left it enabled for now. This default might change once more information is available.

This module is enabled by default. You may use `modules.unload('ta_signal_query')` in your configuration.

8.9 System time skew detector

This module compares local system time with inception and expiration time bounds in DNSSEC signatures for `. NS` records. If the local system time is outside of these bounds, it is likely a misconfiguration which will cause all DNSSEC validation (and resolution) to fail.

In case of mismatch, a warning message will be logged to help with further diagnostics.

> **Warning:** Information printed by this module can be forged by a network attacker! System administrator MUST verify values printed by this module and fix local system time using a trusted source.

This module is useful for debugging purposes. It runs only once during resolver start does not anything after that. It is enabled by default. You may disable the module by appending `modules.unload('detect_time_skew')` to your configuration.

8.10 Detect discontinuous jumps in the system time

This module detect discontinuous jumps in the system time when resolver is running. It clears cache when a significant backward time jumps occurs.

Time jumps are usually created by NTP time change or by admin intervention. These change can affect cache records as they store timestamp and TTL in real time.

If you want to preserve cache during time travel you should disable this module by `modules.unload('detect_time_jump')`.

Due to the way monotonic system time works on typical systems, suspend-resume cycles will be perceived as forward time jumps, but this direction of shift does not have the risk of using records beyond their intended TTL, so forward jumps do not cause erasing the cache.

CHAPTER 9

DNSSEC, data verification

Good news! Knot Resolver uses secure configuration by default, and this configuration should not be changed unless absolutely necessary, so feel free to skip over this section.

Warning: Options in this section are intended only for expert users and normally should not be needed.

Since version 4.0, **DNSSEC validation is enabled by default**. If you really need to turn DNSSEC off and are okay with lowering security of your system by doing so, add the following snippet to your configuration file.

```
-- turns off DNSSEC validation
trust_anchors.remove('.')
```

The resolver supports DNSSEC including **RFC 5011** automated DNSSEC TA updates and **RFC 7646** negative trust anchors. Depending on your distribution, DNSSEC trust anchors should be either maintained in accordance with the distro-wide policy, or automatically maintained by the resolver itself.

In practice this means that you can forget about it and your favorite Linux distribution will take care of it for you.

Following functions allow to modify DNSSEC configuration *if you really have to*:

trust_anchors.add_file (keyfile[, readonly = *false]*)

Parameters

- **keyfile** (*string*) – path to the file.
- **readonly** – if true, do not attempt to update the file.

The format is standard zone file, though additional information may be persisted in comments. Either DS or DNSKEY records can be used for TAs. If the file does not exist, bootstrapping of *root* TA will be attempted. If you want to use bootstrapping, install lua-http library.

Each file can only contain records for a single domain. The TAs will be updated according to **RFC 5011** and persisted in the file (if allowed).

Example output:

```
> trust_anchors.add_file('root.key')
[ ta ] new state of trust anchors for a domain:
.                       165488  DS      19036 8 2␣
↪49AAC11D7B6F6446702E54A1607371607A1A41855200FD2CE1CDDE32F24E8FB5
nil

[ ta ] key: 19036 state: Valid
```

trust_anchors.remove (zonename)
Remove specified trust anchor from trusted key set. Removing trust anchor for the root zone effectivelly disables DNSSEC validation (unless you configured another trust anchor).

```
> trust_anchors.remove('.')
true
```

If you want to disable DNSSEC validation for a particular domain but keep it enabled for the rest of DNS tree, use *trust_anchors.set_insecure()*.

trust_anchors.hold_down_time = 30 * day

Return int (default: 30 * day)

Modify RFC5011 hold-down timer to given value. Intended only for testing purposes. Example: `30 * sec`

trust_anchors.refresh_time = nil

Return int (default: nil)

Modify RFC5011 refresh timer to given value (not set by default), this will force trust anchors to be updated every N seconds periodically instead of relying on RFC5011 logic and TTLs. Intended only for testing purposes. Example: `10 * sec`

trust_anchors.keep_removed = 0

Return int (default: 0)

How many `Removed` keys should be held in history (and key file) before being purged. Note: all `Removed` keys will be purged from key file after restarting the process.

trust_anchors.set_insecure (nta_set)

Parameters

- **nta_list** (*table*) – List of domain names (text format) representing NTAs.

When you use a domain name as an *negative trust anchor* (NTA), DNSSEC validation will be turned off at/below these names. Each function call replaces the previous NTA set. You can find the current active set in `trust_anchors.insecure` variable. If you want to disable DNSSEC validation completely use *trust_anchors.remove()* function instead.

Example output:

```
> trust_anchors.set_insecure({ 'bad.boy', 'example.com' })
> trust_anchors.insecure
[1] => bad.boy
[2] => example.com
```

Warning: If you set NTA on a name that is not a zone cut, it may not always affect names not separated from the NTA by a zone cut.

trust_anchors.add(rr_string)

Parameters

- **rr_string** (*string*) – DS/DNSKEY records in presentation format (e.g. `. 3600 IN DS 19036 8 2 49AAC11...`)

Inserts DS/DNSKEY record(s) into current keyset. These will not be managed or updated, use it only for testing or if you have a specific use case for not using a keyfile.

Note: Static keys are very error-prone and should not be used in production. Use *trust_anchors.add_file()* instead.

Example output:

```
> trust_anchors.add('. 3600 IN DS 19036 8 2 49AAC11...')
```

trust_anchors.summary()

Return string with summary of configured DNSSEC trust anchors, including negative TAs.

DNSSEC is main technology to protect data, but it is also possible to change how strictly resolver checks data from insecure DNS zones:

mode(['strict' | 'normal' | *'permissive']*)

Param New checking level specified as string (*optional*).

Returns Current checking level.

Get or change resolver strictness checking level.

By default, resolver runs in *normal* mode. There are possibly many small adjustments hidden behind the mode settings, but the main idea is that in *permissive* mode, the resolver tries to resolve a name with as few lookups as possible, while in *strict* mode it spends much more effort resolving and checking referral path. However, if majority of the traffic is covered by DNSSEC, some of the strict checking actions are counter-productive.

Glue type	Modes when it is accepted	Example glue[1]
mandatory glue	strict, normal, permissive	ns1.example.org
in-bailiwick glue	normal, permissive	ns1.example2.org
any glue records	permissive	ns1.example3.net

[1] The examples show glue records acceptable from servers authoritative for *org* zone when delegating to *example.org* zone. Unacceptable or missing glue records trigger resolution of names listed in NS records before following respective delegation.

CHAPTER 10

Experimental features

Following functionality and APIs are in continuous development. Features in this section may changed, replaced or dropped in any release.

10.1 Run-time reconfiguration

Knot Resolver offers several ways to modify its configuration at run-time:

- Using control socket driven by an external system
- Using Lua program embeded in Resolver's configuration file

Both ways can also be combined: For example the configuration file can contain a little Lua function which gathers statistics and returns them in JSON string. This can be used by an external system which uses control socket to call this user-defined function and to retrieve its results.

10.1.1 Control sockets

Control socket acts like "an interactive configuration file" so all actions available in configuration file can be executed interactively using the control socket. One possible use-case is reconfiguring Resolver instances from another program, e.g. a maintenance script.

Note: Each instance of Knot Resolver exposes its own control socket. Take that into account when scripting deployments with *Multiple instances*.

When Knot Resolver is started using Systemd (see section *Startup*) it creates a control socket in path `/run/knot-resolver/control/$ID`. Connection to the socket can be made from command line using e.g. `netcat` or `socat`:

```
$ nc -U /run/knot-resolver/control/1
or
$ socat - UNIX-CONNECT:/run/knot-resolver/control/1
```

When successfully connected to a socket, the command line should change to something like >. Then you can interact with kresd to see configuration or set a new one. There are some basic commands to start with.

```
> help()             -- shows help
> net.interfaces()   -- lists available interfaces
> net.list()         -- lists running network services
```

The *direct output* of commands sent over socket is captured and sent back, while also printed to the daemon standard outputs (in `verbose()` mode). This gives you an immediate response on the outcome of your command. Error or debug logs aren't captured, but you can find them in the daemon standard outputs.

Control sockets are also a way to enumerate and test running instances, the list of sockets corresponds to the list of processes, and you can test the process for liveliness by connecting to the UNIX socket.

10.1.2 Lua scripts

As it was mentioned in section *Syntax*, Resolver's configuration file contains program in Lua programming language. This allows you to write dynamic rules and helps you to avoid repetitive templating that is unavoidable with static configuration. For example parts of configuration can depend on `hostname()` of the machine:

```
if hostname() == 'hidden' then
        net.listen(net.eth0, 5353)
else
        net.listen('127.0.0.1')
        net.listen(net.eth1.addr[1])
end
```

Another example would show how it is possible to bind to all interfaces, using iteration.

```
for name, addr_list in pairs(net.interfaces()) do
        net.listen(addr_list)
end
```

Tip: Some users observed a considerable, close to 100%, performance gain in Docker containers when they bound the daemon to a single interface:ip address pair. One may expand the aforementioned example with browsing available addresses as:

```
addrpref = env.EXPECTED_ADDR_PREFIX
for k, v in pairs(addr_list["addr"]) do
        if string.sub(v,1,string.len(addrpref)) == addrpref then
                net.listen(v)
...
```

You can also use third-party Lua libraries (available for example through LuaRocks) as on this example to download cache from parent, to avoid cold-cache start.

```
local http = require('socket.http')
local ltn12 = require('ltn12')
```

(continues on next page)

(continued from previous page)

```
local cache_size = 100*MB
local cache_path = '/var/cache/knot-resolver'
cache.open(cache_size, 'lmdb://' .. cache_path)
if cache.count() == 0 then
        cache.close()
        -- download cache from parent
        http.request {
                url = 'http://parent/data.mdb',
                sink = ltn12.sink.file(io.open(cache_path .. '/data.mdb', 'w'))
        }
        -- reopen cache with 100M limit
        cache.open(cache_size, 'lmdb://' .. cache_path)
end
```

Helper functions

Following built-in functions are useful for scripting:

env (table)
Retrieve environment variables.

Example:

```
env.USER -- equivalent to $USER in shell
```

fromjson (JSONstring)

Returns Lua representation of data in JSON string.

Example:

```
> fromjson('{"key1": "value1", "key2": {"subkey1": 1, "subkey2": 2}}')
[key1] => value1
[key2] => {
    [subkey1] => 1
    [subkey2] => 2
}
```

hostname ([fqdn])

Returns Machine hostname.

If called with a parameter, it will set kresd's internal hostname. If called without a parameter, it will return kresd's internal hostname, or the system's POSIX hostname (see gethostname(2)) if kresd's internal hostname is unset.

This also affects ephemeral (self-signed) certificates generated by kresd for DNS over TLS.

package_version ()

Returns Current package version as string.

Example:

```
> package_version()
2.1.1
```

resolve (name, type[, class = *kres.class.IN*, options = *{}*, finish = *nil*, init = *nil]*)

Parameters

- **name** (*string*) – Query name (e.g. 'com.')
- **type** (*number*) – Query type (e.g. `kres.type.NS`)
- **class** (*number*) – Query class *(optional)* (e.g. `kres.class.IN`)
- **options** (*strings*) – Resolution options (see *kr_qflags*)
- **finish** (*function*) – Callback to be executed when resolution completes (e.g. *function cb (pkt, req) end*). The callback gets a packet containing the final answer and doesn't have to return anything.
- **init** (*function*) – Callback to be executed with the *kr_request* before resolution starts.

Returns boolean, `true` if resolution was started

The function can also be executed with a table of arguments instead. This is useful if you'd like to skip some arguments, for example:

```
resolve {
   name = 'example.com',
   type = kres.type.AAAA,
   init = function (req)
   end,
}
```

Example:

```
-- Send query for root DNSKEY, ignore cache
resolve('.', kres.type.DNSKEY, kres.class.IN, 'NO_CACHE')

-- Query for AAAA record
resolve('example.com', kres.type.AAAA, kres.class.IN, 0,
function (pkt, req)
   -- Check answer RCODE
   if pkt:rcode() == kres.rcode.NOERROR then
      -- Print matching records
      local records = pkt:section(kres.section.ANSWER)
      for i = 1, #records do
         local rr = records[i]
         if rr.type == kres.type.AAAA then
            print ('record:', kres.rr2str(rr))
         end
      end
   else
      print ('rcode: ', pkt:rcode())
   end
end)
```

tojson (object)

Returns JSON text representation of *object*.

Example:

```
> testtable = { key1 = "value1", "key2" = { subkey1 = 1, subkey2 = 2 } }
> tojson(testtable)
{"key1":"value1","key2":{"subkey1":1,"subkey2":2}}
```

10.1.3 Asynchronous events

Lua language used in configuration file allows you to script actions upon various events, for example publish statistics each minute. Following example uses built-in function *event.recurrent()* which calls user-supplied anonymous function:

```
modules.load('stats')

-- log statistics every second
local stat_id = event.recurrent(1 * second, function(evid)
     log(table_print(stats.list()))
end)

-- stop printing statistics after first minute
event.after(1 * minute, function(evid)
     event.cancel(stat_id)
end)
```

Note that each scheduled event is identified by a number valid for the duration of the event, you may use it to cancel the event at any time.

To persist state between two invocations of a fuction Lua uses concept called closures. In the following example function `speed_monitor()` is a closure function, which provides persistent variable called `previous`.

```
modules.load('stats')

-- make a closure, encapsulating counter
function speed_monitor()
    local previous = stats.list()
    -- monitoring function
    return function(evid)
        local now = stats.list()
        local total_increment = now['answer.total'] - previous['answer.total']
        local slow_increment = now['answer.slow'] - previous['answer.slow']
        if slow_increment / total_increment > 0.05 then
            log('WARNING! More than 5 %% of queries was slow!')
        end
        previous = now  -- store current value in closure
     end
 end

 -- monitor every minute
 local monitor_id = event.recurrent(1 * minute, speed_monitor())
```

Another type of actionable event is activity on a file descriptor. This allows you to embed other event loops or monitor open files and then fire a callback when an activity is detected. This allows you to build persistent services like monitoring probes that cooperate well with the daemon internal operations. See *event.socket()*.

Filesystem watchers are possible with *worker.coroutine()* and cqueues, see the cqueues documentation for more information. Here is an simple example:

```
local notify = require('cqueues.notify')
local watcher = notify.opendir('/etc')
watcher:add('hosts')

-- Watch changes to /etc/hosts
worker.coroutine(function ()
  for flags, name in watcher:changes() do
```

(continues on next page)

(continued from previous page)

```
    for flag in notify.flags(flags) do
      -- print information about the modified file
      print(name, notify[flag])
    end
  end
end)
```

Timers and events reference

The timer represents exactly the thing described in the examples - it allows you to execute closures after specified time, or event recurrent events. Time is always described in milliseconds, but there are convenient variables that you can use - `sec, minute, hour`. For example, `5 * hour` represents five hours, or 5*60*60*100 milliseconds.

event.after(time, function)

Returns event id

Execute function after the specified time has passed. The first parameter of the callback is the event itself.

Example:

```
event.after(1 * minute, function() print('Hi!') end)
```

event.recurrent(interval, function)

Returns event id

Similar to *event.after()*, periodically execute function after `interval` passes.

Example:

```
msg_count = 0
event.recurrent(5 * sec, function(e)
   msg_count = msg_count + 1
   print('Hi #'..msg_count)
end)
```

event.reschedule(event_id, timeout)

Reschedule a running event, it has no effect on canceled events. New events may reuse the event_id, so the behaviour is undefined if the function is called after another event is started.

Example:

```
local interval = 1 * minute
event.after(1 * minute, function (ev)
   print('Good morning!')
   -- Halven the interval for each iteration
   interval = interval / 2
   event.reschedule(ev, interval)
end)
```

event.cancel(event_id)

Cancel running event, it has no effect on already canceled events. New events may reuse the event_id, so the behaviour is undefined if the function is called after another event is started.

Example:

```
e = event.after(1 * minute, function() print('Hi!') end)
event.cancel(e)
```

Watch for file descriptor activity. This allows embedding other event loops or simply firing events when a pipe endpoint becomes active. In another words, asynchronous notifications for daemon.

event.socket (fd, cb)

Parameters

- **fd** (*number*) – file descriptor to watch
- **cb** – closure or callback to execute when fd becomes active

Returns event id

Execute function when there is activity on the file descriptor and calls a closure with event id as the first parameter, status as second and number of events as third.

Example:

```
e = event.socket(0, function(e, status, nevents)
   print('activity detected')
end)
e.cancel(e)
```

Asynchronous function execution

The *event* package provides a very basic mean for non-blocking execution - it allows running code when activity on a file descriptor is detected, and when a certain amount of time passes. It doesn't however provide an easy to use abstraction for non-blocking I/O. This is instead exposed through the *worker* package (if *cqueues* Lua package is installed in the system).

worker.coroutine (function)

Start a new coroutine with given function (closure). The function can do I/O or run timers without blocking the main thread. See cqueues for documentation of possible operations and synchronization primitives. The main limitation is that you can't wait for a finish of a coroutine from processing layers, because it's not currently possible to suspend and resume execution of processing layers.

Example:

```
worker.coroutine(function ()
  for i = 0, 10 do
    print('executing', i)
    worker.sleep(1)
  end
end)
```

worker.sleep (seconds)

Pause execution of current function (asynchronously if running inside a worker coroutine).

Example:

```
function async_print(testname, sleep)
        log(testname .. ': system time before sleep' .. tostring(os.time())
        worker.sleep(sleep)  -- other corroutines continue execution now
        log(testname .. ': system time AFTER sleep' .. tostring(os.time())
end
```

(continues on next page)

(continued from previous page)

```
worker.coroutine(function() async_print('call #1', 5) end)
worker.coroutine(function() async_print('call #2', 3) end)
```

Output from this example demonstrates that both calls to function `async_print` were executed asynchronously:

```
call #2: system time before sleep 1578065073
call #1: system time before sleep 1578065073
call #2: system time AFTER  sleep 1578065076
call #1: system time AFTER  sleep 1578065078
```

10.1.4 Etcd support

The *etcd* module connects to etcd peers and watches for configuration changes. By default, the module watches the subtree under `/knot-resolver` directory, but you can change this in the etcd library configuration.

The subtree structure corresponds to the configuration variables in the declarative style.

```
$ etcdctl set /knot-resolvevr/net/127.0.0.1 53
$ etcdctl set /knot-resolver/cache/size 10000000
```

Configures all listening nodes to following configuration:

```
net = { '127.0.0.1' }
cache.size = 10000000
```

Example configuration

```
modules.load('etcd')
etcd.config({
        prefix = '/knot-resolver',
        peer = 'http://127.0.0.1:7001'
})
```

Warning: Work in progress!

Dependencies

- lua-etcd library available in LuaRocks

  ```
  $ luarocks install etcd --from=https://mah0x211.github.io/rocks/
  ```

10.2 Experimental DNS-over-TLS Auto-discovery

This experimental module provides automatic discovery of authoritative servers' supporting DNS-over-TLS. The module uses magic NS names to detect SPKI fingerprint which is very similar to dnscurve mechanism.

Warning: This protocol and module is experimental and can be changed or removed at any time. Use at own risk, security properties were not analyzed!

10.2.1 How it works

The module will look for NS target names formatted as: `dot-{base32(sha256(SPKI))}....`

For instance, Knot Resolver will detect NS names formatted like this

```
example.com NS dot-tpwxmgqdaurcqxqsckxvdq5sty3opxlgcbjj43kumdq62kpqr72a.example.com
```

and automatically discover that example.com NS supports DoT with the base64-encoded SPKI digest of `m+12GgMFIiheEhKvUcOynjbn3WYQUp5tVGDh7Snwj/Q=` and will associate it with the IPs of `dot-tpwxmgqdaurcqxqsckxvdq5sty3opxlgcbjj43kumdq62kpqr72a.example.com`.

In that example, the base32 encoded (no padding) version of the sha256 PIN is `tpwxmgqdaurcqxqsckxvdq5sty3opxlgcbjj43kumdq62kpqr72a`, which when converted to base64 translates to `m+12GgMFIiheEhKvUcOynjbn3WYQUp5tVGDh7Snwj/Q=`.

10.2.2 Generating NS target names

To generate the NS target name, use the following command to generate the base32 encoded string of the SPKI fingerprint:

```
openssl x509 -in /path/to/cert.pem  -pubkey -noout | \
openssl pkey -pubin -outform der | \
openssl dgst -sha256 -binary | \
base32 | tr -d '=' | tr '[:upper:]' '[:lower:]'
tpwxmgqdaurcqxqsckxvdq5sty3opxlgcbjj43kumdq62kpqr72a
```

Then add a target to your NS with: `dot-${b32}.a.example.com`

Finally, map `dot-${b32}.a.example.com` to the right set of IPs.

```
...
...
;; QUESTION SECTION:
;example.com.       IN      NS

;; AUTHORITY SECTION:
example.com. 3600   IN      NS      dot-
↪tpwxmgqdaurcqxqsckxvdq5sty3opxlgcbjj43kumdq62kpqr72a.a.example.com.
example.com. 3600   IN      NS      dot-
↪tpwxmgqdaurcqxqsckxvdq5sty3opxlgcbjj43kumdq62kpqr72a.b.example.com.

;; ADDITIONAL SECTION:
dot-tpwxmgqdaurcqxqsckxvdq5sty3opxlgcbjj43kumdq62kpqr72a.a.example.com. 3600 IN A 192.
↪0.2.1
dot-tpwxmgqdaurcqxqsckxvdq5sty3opxlgcbjj43kumdq62kpqr72a.b.example.com. 3600 IN AAAA␣
↪2001:DB8::1
...
...
```

10.2.3 Example configuration

To enable the module, add this snippet to your config:

```
-- Start an experiment, use with caution
modules.load('experimental_dot_auth')
```

This module requires standard `basexx` Lua library which is typically provided by `lua-basexx` package.

10.2.4 Caveats

The module relies on seeing the reply of the NS query and as such will not work if Knot Resolver uses data from its cache. You may need to delete the cache before starting `kresd` to work around this.

The module also assumes that the NS query answer will return both the NS targets in the Authority section as well as the glue records in the Additional section.

10.2.5 Dependencies

- lua-basexx available in LuaRocks

CHAPTER 11

Usage without systemd

Tip: Our upstream packages use systemd integration, which is the recommended way to run kresd. This section is only relevant if you choose to use kresd without systemd integration.

Knot Resolver is designed to be a single process without the use of threads. While the cache is shared, the individual processes are independent. This approach has several benefits, but it also comes with a few downsides, in particular:

- Without the use of threads or forking (deprecated, see #529), multiple processes aren't managed in any way by kresd.
- There is no maintenance thread and these tasks have to be handled by separate daemon(s) (such as *Garbage Collector*).

To offset these these disadvantages without implementing process management in kresd (and reinventing the wheel), Knot Resolver provides integration with systemd, which is widely used across GNU/Linux distributions.

If your use-case doesn't support systemd (e.g. using macOS, FreeBSD, Docker, OpenWrt, Turris), this section describes the differences and things to keep in mind when configuring and running kresd without systemd integration.

11.1 Process management

There following should be taken into consideration when running without systemd:

- To utilize multiple CPUs, kresd has to be executed as several independent processes.
- Maintenance daemon(s) have to be executed separately.
- If a process crashes, it might be useful to restart it.
- Using some mechanism similar to *Watchdog* might be desirable to recover in case a process becomes unresponsive.

Please note, systemd isn't the only process manager and other solutions exist, such as supervisord. Configuring these is out of the scope of this document. Please refer to their respective documentations.

It is also possible to use kresd without any process management at all, which may be suitable for some purposes (such as low-traffic local / home network resolver, testing, development or debugging).

11.1.1 Garbage Collector

Note: When using systemd, `kres-cache-gc.service` is enabled by default and does not need any manual configuration.

Knot Resolver employs a separate garbage collector daemon which periodically trims the cache to keep its size below size limit configured using *`cache.size`*.

To execute the daemon manually, you can use the following command to run it every second:

```
$ kres-cache-gc -c /var/cache/knot-resolver -d 1000
```

11.2 Privileges and capabilities

The kresd daemon requires privileges when it is configured to bind to well-known ports. There are multiple ways to achieve this.

11.2.1 Using capabilities

The most secure and recommended way is to use capabilities and execute kresd as an unprivileged user.

- `CAP_NET_BIND_SERVICE` is required to bind to well-known ports.
- `CAP_SETPCAP` when this capability is available, kresd drops any extra privileges after the daemon successfully starts.

11.2.2 Running as non-privileged user

Another possibility is to start the process as privileged user and then switch to a non-privileged user after binding to network interfaces.

`user`(name, [group])

Parameters

- **`name`** (*`string`*) – user name
- **`group`** (*`string`*) – group name (optional)

Returns boolean

Drop privileges and start running as given user (and group, if provided).

Tip: Note that you should bind to required network addresses before changing user. At the same time, you should open the cache **AFTER** you change the user (so it remains accessible). A good practice is to divide configuration in two parts:

```
-- privileged
net.listen('127.0.0.1')
net.listen('::1')
user('knot-resolver', 'netgrp')
-- unprivileged
cache.size = 100*MB
```

Example output:

```
> user('baduser')
invalid user name
> user('knot-resolver', 'netgrp')
true
> user('root')
Operation not permitted
```

11.2.3 Running as root

Warning: Executing processes as root is generally insecure, as these proccesses have unconstrained access to the complete system at runtime.

While not recommended, it is also possible to run kresd directly as root.

Please note the process will still attempt to drop capabilities after startup. Among other things, this means the cache directory should belong to root to have write access.

CHAPTER 12

Upgrading

This section summarizes steps required for upgrade to newer Knot Resolver versions. We advise users to also read *Release notes* for respective versions.

12.1 4.x to 5.x

12.1.1 Users

- Control socket location has changed

	4.x location	5.x location
with systemd	`/run/knot-resolver/control@$ID`	`/run/knot-resolver/control/$ID`
without systemd	`$PWD/tty/$PID`	`$PWD/control/$PID`

- `-f` / `--forks` command-line option is deprecated. In case you just want to trigger non-interactive mode, there's new `-n` / `--noninteractive`. This forking style was not ergonomic; with independent kresd processes you can better utilize a process manager (e.g. systemd).

12.1.2 Configuration file

- Network interface are now configured in `kresd.conf` with *`net.listen()`* instead of systemd sockets (#485). See the following examples.

Tip: You can find suggested network interface settings based on your previous systemd socket configuration in `/var/lib/knot-resolver/.upgrade-4-to-5/kresd.conf.net` which is created during the package update to version 5.x.

4.x - systemd socket file	5.x - kresd.conf
kresd.socket [Socket] ListenDatagram=127.0.0.1:53 ListenStream=127.0.0.1:53	`net.listen('127.0.0.1', 53, { kind = 'dns' })`
kresd.socket [Socket] FreeBind=true BindIPv6Only=both ListenDatagram=[::1]:53 ListenStream=[::1]:53	`net.listen('127.0.0.1', 53, { kind = 'dns', freebind = true })` `net.listen('::1', 53, { kind = 'dns', freebind = true })`
kresd-tls.socket [Socket] ListenStream=127.0.0.1:853	`net.listen('127.0.0.1', 853, { kind = 'tls' })`
kresd-doh.socket [Socket] ListenStream=127.0.0.1:443	`net.listen('127.0.0.1', 443, { kind = 'doh' })`
kresd-webmgmt.socket [Socket] ListenStream=127.0.0.1:8453	`net.listen('127.0.0.1', 8453, { kind = 'webmgmt' })`

- *`net.listen()`* throws an error if it fails to bind. Use `freebind=true` option to bind to nonlocal addresses.

12.2 4.2.2 to 4.3+

12.2.1 Module changes

- In case you wrote your own module which directly calls function `kr_ranked_rrarray_add()`, you need to additionally call function `kr_ranked_rrarray_finalize()` after each batch (before changing the added memory regions). For a specific example see changes in dns64 module.

12.3 4.x to 4.2.1+

12.3.1 Users

- If you have previously installed `knot-resolver-dbgsym` package on Debian, please remove it and install `knot-resolver-dbg` instead.

12.4 3.x to 4.x

12.4.1 Users

- DNSSEC validation is now turned on by default. If you need to disable it, see *DNSSEC, data verification*.
- `-k/--keyfile` and `-K/--keyfile-ro` daemon options were removed. If needed, use `trust_anchors.add_file()` in configuration file instead.
- Configuration for *HTTP module* changed significantly as result of adding *DNS-over-HTTP (DoH)* support. Please see examples below.
- In case you are using your own custom modules, move them to the new module location. The exact location depends on your distribution. Generally, modules previously in `/usr/lib/kdns_modules` should be moved to `/usr/lib/knot-resolver/kres_modules`.

Configuration file

- `trust_anchors.file`, `trust_anchors.config()` and `trust_anchors.negative` aliases were removed to avoid duplicity and confusion. Migration table:

3.x configuration	4.x configuration
`trust_anchors.file = path`	`trust_anchors.add_file(path)`
`trust_anchors.config(path, readonly)`	`trust_anchors.add_file(path, readonly)`
`trust_anchors.negative = nta_set`	`trust_anchors.set_insecure(nta_set)`

- `trust_anchors.keyfile_default` is no longer accessible and is can be set only at compile time. To turn off DNSSEC, use *`trust_anchors.remove()`*.

3.x configuration	4.x configuration
`trust_anchors.keyfile_default = nil`	`trust_anchors.remove('.')`

- Network for HTTP endpoints is now configured using same mechanism as for normal DNS enpoints, please refer to chapter *Networking and protocols*. Migration table:

3.x configuration	4.x configuration
`modules = { http = { host = '192.0.2.1', port = 443 }}`	see chapter *Networking and protocols*
`http.config({ host = '192.0.2.1', port = 443 })`	see chapter *Networking and protocols*
`modules = { http = { endpoints = ... }}`	see chapter *Custom HTTP services*
`http.config({ endpoints = ... })`	see chapter *Custom HTTP services*

12.4.2 Packagers & Developers

- Knot DNS >= 2.8 is required.
- meson >= 0.46 and ninja is required.

- meson build system is now used for compiling the project. For instructions, see the *Building from sources*. Packagers should pay attention to section *Packaging* for information about systemd unit files and trust anchors.
- Embedding LMDB is no longer supported, lmdb is now required as an external dependency.
- Trust anchors file from upstream is installed and used as default unless you override `keyfile_default` during build.

Module changes

- Default module location has changed from `{libdir}/kdns_modules` to `{libdir}/knot-resolver/kres_modules`. Modules are now in the lua namespace `kres_modules.*`.
- `kr_straddr_split()` API has changed.
- C modules defining `*_layer` or `*_props` symbols need to use a different style, but it's typically a trivial change. Instead of exporting the corresponding symbols, the module should assign pointers to its static structures inside its `*_init()` function. Example migration: bogus_log module.

12.5 2.x to 3.x

12.5.1 Users

- Module *Static hints* has option *`hints.use_nodata()`* enabled by default, which is what most users expect. Add `hints.use_nodata(false)` to your config to revert to the old behavior.
- Modules `cookie` and `version` were removed. Please remove relevant configuration lines with `modules.load()` and `modules =` from configuration file.
- Valid configuration must open cache using `cache.open()` or `cache.size =` before executing cache operations like `cache.clear()`. (Older versions were silently ignoring such cache operations.)

12.5.2 Packagers & Developers

- Knot DNS >= 2.7.2 is required.

Module changes

- API for Lua modules was refactored, please see *Significant Lua API changes*.
- New layer was added: `answer_finalize`.
- `kr_request` keeps `::qsource.packet` beyond the `begin` layer.
- `kr_request::qsource.tcp` renamed to `::qsource.flags.tcp`.
- `kr_request::has_tls` renamed to `::qsource.flags.tls`.
- `kr_zonecut_add()`, `kr_zonecut_del()` and `kr_nsrep_sort()` changed parameters slightly.

CHAPTER 13

Release notes

13.1 Knot Resolver 5.0.0 (2020-01-27)

13.1.1 Incompatible changes

- see upgrading guide: https://knot-resolver.readthedocs.io/en/stable/upgrading.html
- systemd sockets are no longer supported (#485)
- net.listen() throws an error if it fails to bind; use freebind option if needed
- control socket location has changed (!922)
- -f/–forks is deprecated (#529, !919)

13.1.2 Improvements

- logging: control-socket commands don't log unless –verbose (#528)
- use SO_REUSEPORT_LB if available (FreeBSD 12.0+)
- lua: remove dependency on lua-socket and lua-sec, used lua-http and cqueues (#512, #521, !894)
- lua: remove dependency on lua-filesystem (#520, !912)
- net.listen(): allow binding to non-local address with freebind option (!898)
- cache: pre-allocate the file to avoid SIGBUS later (not macOS; !917, #525)
- lua: be stricter around nonsense returned from modules (!901)
- user documentation was reorganized and extended (!900, !867)
- multiple config files can be used with –config/-c option (!909)
- lua: stop trying to tweak lua's GC (!201)
- systemd: add SYSTEMD_INSTANCE env variable to identify different instances (!906)

13.1.3 Bugfixes

- correctly use EDNS(0) padding in failed answers (!921)
- policy and daf modules: fix postrules and reroute rules (!901)
- renumber module: don't accidentally zero-out request's .state (!901)

13.2 Knot Resolver 4.3.0 (2019-12-04)

13.2.1 Security - CVE-2019-19331

- fix speed of processing large RRsets (DoS, #518)
- improve CNAME chain length accounting (DoS, !899)

13.2.2 Bugfixes

- http module: use SO_REUSEPORT (!879)
- systemd: kresd@.service now properly starts after network interfaces have been configured with IP addresses after reboot (!884)
- sendmmsg: improve reliability (!704)
- cache: fix crash on insertion via lua for NS and CNAME (!889)
- rpm package: move root.keys to /var/lib/knot-resolver (#513, !888)

13.2.3 Improvements

- increase file-descriptor count limit to maximum allowed value (hard limit; !876)
- watchdog module: support testing a DNS query (and switch C -> lua; !878, !881)
- performance: use sendmmsg syscall towards clients by default (!877)
- performance: avoid excessive getsockname() syscalls (!854)
- performance: lua-related improvements (!874)
- daemon now attempts to drop all capabilities (!896)
- reduce CNAME chain length limit - now <= 12 (!899)

13.3 Knot Resolver 4.2.2 (2019-10-07)

13.3.1 Bugfixes

- lua bindings: fix a 4.2.1 regression on 32-bit systems (#514) which also fixes libknot 2.9 support on all systems

13.4 Knot Resolver 4.2.1 (2019-09-26)

13.4.1 Bugfixes

- rebinding module: fix handling some requests, respect ALLOW_LOCAL flag
- fix incorrect SERVFAIL on cached bogus answer for +cd request (!860) (regression since 4.1.0 release, in less common cases)
- prefill module: allow a different module-loading style (#506)
- validation: trim TTLs by RRSIG's expiration and original TTL (#319, #504)
- NS choice algorithm: fix a regression since 4.0.0 (#497, !868)
- policy: special domains home.arpa. and local. get NXDOMAIN (!855)

13.4.2 Improvements

- add compatibility with (future) libknot 2.9

13.5 Knot Resolver 4.2.0 (2019-08-05)

13.5.1 Improvements

- queries without RD bit set are REFUSED by default (!838)
- support forwarding to multiple targets (!825)

13.5.2 Bugfixes

- tls_client: fix issue with TLS session resumption (#489)
- rebinding module: fix another false-positive assertion case (!851)

13.5.3 Module API changes

- kr_request::add_selected is now really put into answer, instead of the "duplicate" ::additional field (#490)

13.6 Knot Resolver 4.1.0 (2019-07-10)

13.6.1 Security

- fix CVE-2019-10190: do not pass bogus negative answer to client (!827)
- fix CVE-2019-10191: do not cache negative answer with forged QNAME+QTYPE (!839)

13.6.2 Improvements

- new cache garbage collector is available and enabled by default (#257) This improves cache efficiency on big installations.
- DNS-over-HTTPS: unknown HTTP parameters are ignored to improve compatibility with non-standard clients (!832)
- DNS-over-HTTPS: answers include *access-control-allow-origin: ** (!823) which allows JavaScript to use DoH endpoint.
- http module: support named AF_UNIX stream sockets (again)
- aggressive caching is disabled on minimal NSEC* ranges (!826) This improves cache effectivity with DNSSEC black lies and also accidentally works around bug in proofs-of-nonexistence from F5 BIG-IP load-balancers.
- aarch64 support, even kernels with ARM64_VA_BITS >= 48 (#216, !797) This is done by working around a LuaJIT incompatibility. Please report bugs.
- lua tables for C modules are more strict by default, e.g. *nsid.foo* will throw an error instead of returning *nil* (!797)
- systemd: basic watchdog is now available and enabled by default (#275)

13.6.3 Bugfixes

- TCP to upstream: fix unlikely case of sending out wrong message length (!816)
- http module: fix problems around maintenance of ephemeral certs (!819)
- http module: also send intermediate TLS certificate to clients, if available and luaossl >= 20181207 (!819)
- send EDNS with SERVFAILs, e.g. on validation failures (#180, !827)
- prefill module: avoid crash on empty zone file (#474, !840)
- rebinding module: avoid excessive iteration on blocked attempts (!842)
- rebinding module: fix crash caused by race condition (!842)
- rebinding module: log each blocked query only in verbose mode (!842)
- cache: automatically clear stale reader locks (!844)

13.6.4 Module API changes

- lua modules may omit casting parameters of layer functions (!797)

13.7 Knot Resolver 4.0.0 (2019-04-18)

13.7.1 Incompatible changes

- see upgrading guide: https://knot-resolver.readthedocs.io/en/stable/upgrading.html
- configuration: trust_anchors aliases .file, .config() and .negative were removed (!788)
- configuration: trust_anchors.keyfile_default is no longer accessible (!788)
- daemon: -k/–keyfile and -K/–keyfile-ro options were removed

- meson build system is now used for builds (!771)
- build with embedded LMBD is no longer supported
- default modules dir location has changed
- DNSSEC is enabled by default
- upstream packages for Debian now require systemd
- libknot >= 2.8 is required
- net.list() output format changed (#448)
- net.listen() reports error when address-port pair is in use
- bind to DNS-over-TLS port by default (!792)
- stop versioning libkres library
- default port for web management and APIs changed to 8453

13.7.2 Improvements

- policy.TLS_FORWARD: if hostname is configured, send it on wire (!762)
- hints module: allow configuring the TTL and change default from 0 to 5s
- policy module: policy.rpz() will watch the file for changes by default
- packaging: lua cqueues added to default dependencies where available
- systemd: service is no longer auto-restarted on configuration errors
- always send DO+CD flags upstream, even in insecure zones (#153)
- cache.stats() output is completely new; see docs (!775)
- improve usability of table_print() (!790, !801)
- add DNS-over-HTTPS support (#280)
- docker image supports and exposes DNS-over-HTTPS

13.7.3 Bugfixes

- predict module: load stats module if config didn't specify period (!755)
- trust_anchors: don't do 5011-style updates on anchors from files that were loaded as unmanaged trust anchors (!753)
- trust_anchors.add(): include these TAs in .summary() (!753)
- policy module: support '#' for separating port numbers, for consistency
- fix startup on macOS+BSD when </dev/null and cqueues installed
- policy.RPZ: log problems from zone-file level of parser as well (#453)
- fix flushing of messages to logs in some cases (notably systemd) (!781)
- fix fallback when SERVFAIL or REFUSED is received from upstream (!784)
- fix crash when dealing with unknown TA key algorithm (#449)
- go insecure due to algorithm support even if DNSKEY is NODATA (!798)

- fix mac addresses in the output of net.interfaces() command (!804)
- http module: fix too early renewal of ephemeral certificates (!808)

13.7.4 Module API changes

- kr_straddr_split() changed API a bit (compiler will catch that)
- C modules defining **_layer* or **_props* symbols need to change a bit See the upgrading guide for details. It's detected on module load.

13.8 Knot Resolver 3.2.1 (2019-01-10)

13.8.1 Bugfixes

- trust_anchors: respect validity time range during TA bootstrap (!748)
- fix TLS rehandshake handling (!739)
- make TLS_FORWARD compatible with GnuTLS 3.3 (!741)
- special thanks to Grigorii Demidov for his long-term work on Knot Resolver!

13.8.2 Improvements

- improve handling of timeouted outgoing TCP connections (!734)
- trust_anchors: check syntax of public keys in DNSKEY RRs (!748)
- validator: clarify message about bogus non-authoritative data (!735)
- dnssec validation failures contain more verbose reasoning (!735)
- new function trust_anchors.summary() describes state of DNSSEC TAs (!737), and logs new state of trust anchors after start up and automatic changes
- trust anchors: refuse revoked DNSKEY even if specified explicitly, and downgrade missing the SEP bit to a warning

13.9 Knot Resolver 3.2.0 (2018-12-17)

13.9.1 New features

- module edns_keepalive to implement server side of RFC 7828 (#408)
- module nsid to implement server side of RFC 5001 (#289)
- module bogus_log provides .frequent() table (!629, credit Ulrich Wisser)
- module stats collects flags from answer messages (!629, credit Ulrich Wisser)
- module view supports multiple rules with identical address/TSIG specification and keeps trying rules until a "non-chain" action is executed (!678)
- module experimental_dot_auth implements an DNS-over-TLS to auth protocol (!711, credit Manu Bretelle)

- net.bpf bindings allow advanced users to use eBPF socket filters

13.9.2 Bugfixes

- http module: only run prometheus in parent process if using –forks=N, as the submodule collects metrics from all sub-processes as well.
- TLS fixes for corner cases (!700, !714, !716, !721, !728)
- fix build with -DNOVERBOSELOG (#424)
- policy.{FORWARD,TLS_FORWARD,STUB}: respect net.ipv{4,6} setting (!710)
- avoid SERVFAILs due to certain kind of NS dependency cycles, again (#374) this time seen as 'circular dependency' in verbose logs
- policy and view modules do not overwrite result finished requests (!678)

13.9.3 Improvements

- Dockerfile: rework, basing on Debian instead of Alpine
- policy.{FORWARD,TLS_FORWARD,STUB}: give advantage to IPv6 when choosing whom to ask, just as for iteration
- use pseudo-randomness from gnutls instead of internal ISAAC (#233)
- tune the way we deal with non-responsive servers (!716, !723)
- documentation clarifies interaction between policy and view modules (!678, !730)

13.9.4 Module API changes

- new layer is added: answer_finalize
- kr_request keeps ::qsource.packet beyond the begin layer
- kr_request::qsource.tcp renamed to ::qsource.flags.tcp
- kr_request::has_tls renamed to ::qsource.flags.tls
- kr_zonecut_add(), kr_zonecut_del() and kr_nsrep_sort() changed parameters slightly

13.10 Knot Resolver 3.1.0 (2018-11-02)

13.10.1 Incompatible changes

- hints.use_nodata(true) by default; that's what most users want
- libknot >= 2.7.2 is required

13.10.2 Improvements

- cache: handle out-of-space SIGBUS slightly better (#197)
- daemon: improve TCP timeout handling (!686)

13.10.3 Bugfixes

- cache.clear('name'): fix some edge cases in API (#401)
- fix error handling from TLS writes (!669)
- avoid SERVFAILs due to certain kind of NS dependency cycles (#374)

13.11 Knot Resolver 3.0.0 (2018-08-20)

13.11.1 Incompatible changes

- cache: fail lua operations if cache isn't open yet (!639) By default cache is opened *after* reading the configuration, and older versions were silently ignoring cache operations. Valid configuration must open cache using *cache.open()* or *cache.size* = before executing cache operations like *cache.clear()*.
- libknot >= 2.7.1 is required, which brings also larger API changes
- in case you wrote custom Lua modules, please consult https://knot-resolver.readthedocs.io/en/latest/lib.html#incompatible-changes-since-3-0-0
- in case you wrote custom C modules, please see compile against Knot DNS 2.7 and adjust your module according to messages from C compiler
- DNS cookie module (RFC 7873) is not available in this release, it will be later reworked to reflect development in IEFT dnsop working group
- version module was permanently removed because it was not really used by users; if you want to receive notifications abou new releases please subscribe to https://lists.nic.cz/cgi-bin/mailman/listinfo/knot-resolver-announce

13.11.2 Bugfixes

- fix multi-process race condition in trust anchor maintenance (!643)
- ta_sentinel: also consider static trust anchors not managed via RFC 5011

13.11.3 Improvements

- reorder_RR() implementation is brought back
- bring in performace improvements provided by libknot 2.7
- cache.clear() has a new, more powerful API
- cache documentation was improved
- old name "Knot DNS Resolver" is replaced by unambiguous "Knot Resolver" to prevent confusion with "Knot DNS" authoritative server

13.12 Knot Resolver 2.4.1 (2018-08-02)

13.12.1 Security

- fix CVE-2018-10920: Improper input validation bug in DNS resolver component (security!7, security!9)

13.12.2 Bugfixes

- cache: fix TTL overflow in packet due to min_ttl (#388, security!8)
- TLS session resumption: avoid bad scheduling of rotation (#385)
- HTTP module: fix a regression in 2.4.0 which broke custom certs (!632)
- cache: NSEC3 negative cache even without NS record (#384) This fixes lower hit rate in NSEC3 zones (since 2.4.0).
- minor TCP and TLS fixes (!623, !624, !626)

13.13 Knot Resolver 2.4.0 (2018-07-03)

13.13.1 Incompatible changes

- minimal libknot version is now 2.6.7 to pull in latest fixes (#366)

13.13.2 Security

- fix a rare case of zones incorrectly dowgraded to insecure status (!576)

13.13.3 New features

- TLS session resumption (RFC 5077), both server and client (!585, #105) (disabled when compiling with gnutls < 3.5)
- TLS_FORWARD policy uses system CA certificate store by default (!568)
- aggressive caching for NSEC3 zones (!600)
- optional protection from DNS Rebinding attack (module rebinding, !608)
- module bogus_log to log DNSSEC bogus queries without verbose logging (!613)

13.13.4 Bugfixes

- prefill: fix ability to read certificate bundle (!578)
- avoid turning off qname minimization in some cases, e.g. co.uk. (#339)
- fix validation of explicit wildcard queries (#274)
- dns64 module: more properties from the RFC implemented (incl. bug #375)

13.13.5 Improvements

- systemd: multiple enabled kresd instances can now be started using kresd.target
- ta_sentinel: switch to version 14 of the RFC draft (!596)
- support for glibc systems with a non-Linux kernel (!588)
- support per-request variables for Lua modules (!533)

- support custom HTTP endpoints for Lua modules (!527)

13.14 Knot Resolver 2.3.0 (2018-04-23)

13.14.1 Security

- fix CVE-2018-1110: denial of service triggered by malformed DNS messages (!550, !558, security!2, security!4)
- increase resilience against slow lorris attack (security!5)

13.14.2 New features

- new policy.REFUSE to reply REFUSED to clients

13.14.3 Bugfixes

- validation: fix SERVFAIL in case of CNAME to NXDOMAIN in a single zone (!538)
- validation: fix SERVFAIL for DS . query (!544)
- lib/resolve: don't send unecessary queries to parent zone (!513)
- iterate: fix validation for zones where parent and child share NS (!543)
- TLS: improve error handling and documentation (!536, !555, !559)

13.14.4 Improvements

- prefill: new module to periodically import root zone into cache (replacement for RFC 7706, !511)
- network_listen_fd: always create end point for supervisor supplied file descriptor
- use CPPFLAGS build environment variable if set (!547)

13.15 Knot Resolver 2.2.0 (2018-03-28)

13.15.1 New features

- cache server unavailability to prevent flooding unreachable servers (Please note that caching algorithm needs further optimization and will change in further versions but we need to gather operational experience first.)

13.15.2 Bugfixes

- don't magically -D_FORTIFY_SOURCE=2 in some cases
- allow large responses for outbound over TCP
- fix crash with RR sets with over 255 records

13.16 Knot Resolver 2.1.1 (2018-02-23)

13.16.1 Bugfixes

- when iterating, avoid unnecessary queries for NS in insecure parent. This problem worsened in 2.0.0. (#246)
- prevent UDP packet leaks when using TLS forwarding
- fix the hints module also on some other systems, e.g. Gentoo.

13.17 Knot Resolver 2.1.0 (2018-02-16)

13.17.1 Incompatible changes

- stats: remove tracking of expiring records (predict uses another way)
- systemd: re-use a single kresd.socket and kresd-tls.socket
- ta_sentinel: implement protocol draft-ietf-dnsop-kskroll-sentinel-01 (our draft-ietf-dnsop-kskroll-sentinel-00 implementation had inverted logic)
- libknot: require version 2.6.4 or newer to get bugfixes for DNS-over-TLS

13.17.2 Bugfixes

- detect_time_jump module: don't clear cache on suspend-resume (#284)
- stats module: fix stats.list() returning nothing, regressed in 2.0.0
- policy.TLS_FORWARD: refusal when configuring with multiple IPs (#306)
- cache: fix broken refresh of insecure records that were about to expire
- fix the hints module on some systems, e.g. Fedora (came back on 2.0.0)
- build with older gnutls (conditionally disable features)
- fix the predict module to work with insecure records & cleanup code

13.18 Knot Resolver 2.0.0 (2018-01-31)

13.18.1 Incompatible changes

- systemd: change unit files to allow running multiple instances, deployments with single instance now must use *kresd@1.service* instead of *kresd.service*; see kresd.systemd(7) for details
- systemd: the directory for cache is now /var/cache/knot-resolver
- unify default directory and user to *knot-resolver*
- directory with trust anchor file specified by -k option must be writeable
- policy module is now loaded by default to enforce RFC 6761; see documentation for policy.PASS if you use locally-served DNS zones

- drop support for alternative cache backends memcached, redis, and for Lua bindings for some specific cache operations
- REORDER_RR option is not implemented (temporarily)

13.18.2 New features

- aggressive caching of validated records (RFC 8198) for NSEC zones; thanks to ICANN for sponsoring this work.
- forwarding over TLS, authenticated by SPKI pin or certificate. policy.TLS_FORWARD pipelines queries out-of-order over shared TLS connection Beware: Some resolvers do not support out-of-order query processing. TLS forwarding to such resolvers will lead to slower resolution or failures.
- trust anchors: you may specify a read-only file via -K or –keyfile-ro
- trust anchors: at build-time you may set KEYFILE_DEFAULT (read-only)
- ta_sentinel module implements draft ietf-dnsop-kskroll-sentinel-00, enabled by default
- serve_stale module is prototype, subject to change
- extended API for Lua modules

13.18.3 Bugfixes

- fix build on osx - regressed in 1.5.3 (different linker option name)

13.19 Knot Resolver 1.5.3 (2018-01-23)

13.19.1 Bugfixes

- fix the hints module on some systems, e.g. Fedora. Symptom: *undefined symbol: engine_hint_root_file*

13.20 Knot Resolver 1.5.2 (2018-01-22)

13.20.1 Security

- fix CVE-2018-1000002: insufficient DNSSEC validation, allowing attackers to deny existence of some data by forging packets. Some combinations pointed out in RFC 6840 sections 4.1 and 4.3 were not taken into account.

13.20.2 Bugfixes

- memcached: fix fallout from module rename in 1.5.1

13.21 Knot Resolver 1.5.1 (2017-12-12)

13.21.1 Incompatible changes

- script supervisor.py was removed, please migrate to a real process manager
- module ketcd was renamed to etcd for consistency
- module kmemcached was renamed to memcached for consistency

13.21.2 Bugfixes

- fix SIGPIPE crashes (#271)
- tests: work around out-of-space for platforms with larger memory pages
- lua: fix mistakes in bindings affecting 1.4.0 and 1.5.0 (and 1.99.1-alpha), potentially causing problems in dns64 and workarounds modules
- predict module: various fixes (!399)

13.21.3 Improvements

- add priming module to implement RFC 8109, enabled by default (#220)
- add modules helping with system time problems, enabled by default; for details see documentation of detect_time_skew and detect_time_jump

13.22 Knot Resolver 1.5.0 (2017-11-02)

13.22.1 Bugfixes

- fix loading modules on Darwin

13.22.2 Improvements

- new module ta_signal_query supporting Signaling Trust Anchor Knowledge using Keytag Query (RFC 8145 section 5); it is enabled by default
- attempt validation for more records but require it for fewer of them (e.g. avoids SERVFAIL when server adds extra records but omits RRSIGs)

13.23 Knot Resolver 1.99.1-alpha (2017-10-26)

This is an experimental release meant for testing aggressive caching. It contains some regressions and might (theoretically) be even vulnerable. The current focus is to minimize queries into the root zone.

13.23.1 Improvements

- negative answers from validated NSEC (NXDOMAIN, NODATA)
- verbose log is very chatty around cache operations (maybe too much)

13.23.2 Regressions

- dropped support for alternative cache backends and for some specific cache operations
- **caching doesn't yet work for various cases:**
  - **negative answers without NSEC (i.e. with NSEC3 or insecure)**
    - +cd queries (needs other internal changes)
    - positive wildcard answers
- **spurious SERVFAIL on specific combinations of cached records, printing:** <= bad keys, broken trust chain
- make check
- a few Deckard tests are broken, probably due to some problems above
- also unknown ones?

13.24 Knot Resolver 1.4.0 (2017-09-22)

13.24.1 Incompatible changes

- lua: query flag-sets are no longer represented as plain integers. kres.query.* no longer works, and kr_query_t lost trivial methods 'hasflag' and 'resolved'. You can instead write code like qry.flags.NO_0X20 = true.

13.24.2 Bugfixes

- fix exiting one of multiple forks (#150)
- cache: change the way of using LMDB transactions. That in particular fixes some cases of using too much space with multiple kresd forks (#240).

13.24.3 Improvements

- policy.suffix: update the aho-corasick code (#200)
- root hints are now loaded from a zonefile; exposed as hints.root_file(). You can override the path by defining ROOTHINTS during compilation.
- policy.FORWARD: work around resolvers adding unsigned NS records (#248)
- reduce unneeded records previously put into authority in wildcarded answers

13.25 Knot Resolver 1.3.3 (2017-08-09)

13.25.1 Security

- Fix a critical DNSSEC flaw. Signatures might be accepted as valid even if the signed data was not in bailiwick of the DNSKEY used to sign it, assuming the trust chain to that DNSKEY was valid.

13.25.2 Bugfixes

- iterate: skip RRSIGs with bad label count instead of immediate SERVFAIL
- utils: fix possible incorrect seeding of the random generator
- modules/http: fix compatibility with the Prometheus text format

13.25.3 Improvements

- policy: implement remaining special-use domain names from RFC6761 (#205), and make these rules apply only if no other non-chain rule applies

13.26 Knot Resolver 1.3.2 (2017-07-28)

13.26.1 Security

- fix possible opportunities to use insecure data from cache as keys for validation

13.26.2 Bugfixes

- daemon: check existence of config file even if rundir isn't specified
- policy.FORWARD and STUB: use RTT tracking to choose servers (#125, #208)
- dns64: fix CNAME problems (#203) It still won't work with policy.STUB.
- **hints: better interpretation of hosts-like files (#204)** also, error out if a bad entry is encountered in the file
- dnssec: handle unknown DNSKEY/DS algorithms (#210)
- predict: fix the module, broken since 1.2.0 (#154)

13.26.3 Improvements

- embedded LMDB fallback: update 0.9.18 -> 0.9.21

13.27 Knot Resolver 1.3.1 (2017-06-23)

13.27.1 Bugfixes

- modules/http: fix finding the static files (bug from 1.3.0)

- policy.FORWARD: fix some cases of CNAMEs obstructing search for zone cuts

13.28 Knot Resolver 1.3.0 (2017-06-13)

13.28.1 Security

- Refactor handling of AD flag and security status of resource records. In some cases it was possible for secure domains to get cached as insecure, even for a TLD, leading to disabled validation. It also fixes answering with non-authoritative data about nameservers.

13.28.2 Improvements

- major feature: support for forwarding with validation (#112). The old policy.FORWARD action now does that; the previous non-validating mode is still available as policy.STUB except that also uses caching (#122).
- command line: specify ports via @ but still support # for compatibility
- policy: recognize 100.64.0.0/10 as local addresses
- layer/iterate: *do* retry repeatedly if REFUSED, as we can't yet easily retry with other NSs while avoiding retrying with those who REFUSED
- modules: allow changing the directory where modules are found, and do not search the default library path anymore.

13.28.3 Bugfixes

- validate: fix insufficient caching for some cases (relatively rare)
- avoid putting "duplicate" record-sets into the answer (#198)

13.29 Knot Resolver 1.2.6 (2017-04-24)

13.29.1 Security

- dnssec: don't set AD flag for NODATA answers if wildcard non-existence is not guaranteed due to opt-out in NSEC3

13.29.2 Improvements

- layer/iterate: don't retry repeatedly if REFUSED

13.29.3 Bugfixes

- lib/nsrep: revert some changes to NS reputation tracking that caused severe problems to some users of 1.2.5 (#178 and #179)
- dnssec: fix verification of wildcarded non-singleton RRsets
- dnssec: allow wildcards located directly under the root

- layer/rrcache: avoid putting answer records into queries in some cases

13.30 Knot Resolver 1.2.5 (2017-04-05)

13.30.1 Security

- layer/validate: clear AD if closest encloser proof has opt-outed NSEC3 (#169)
- layer/validate: check if NSEC3 records in wildcard expansion proof has an opt-out
- dnssec/nsec: missed wildcard no-data answers validation has been implemented

13.30.2 Improvements

- modules/dnstap: a DNSTAP support module (Contributed by Vicky Shrestha)
- modules/workarounds: a module adding workarounds for known DNS protocol violators
- layer/iterate: fix logging of glue addresses
- kr_bitcmp: allow bits=0 and consequently 0.0.0.0/0 matches in view and renumber modules.
- modules/padding: Improve default padding of responses (Contributed by Daniel Kahn Gillmor)
- New kresc client utility (experimental; don't rely on the API yet)

13.30.3 Bugfixes

- trust anchors: Improve trust anchors storage format (#167)
- trust anchors: support non-root TAs, one domain per file
- policy.DENY: set AA flag and clear AD flag
- lib/resolve: avoid unnecessary DS queries
- lib/nsrep: don't treat servers with NOIP4 + NOIP6 flags as timeouted
- layer/iterate: During packet classification (answer vs. referral) don't analyze AUTHORITY section in authoritative answer if ANSWER section contains records that have been requested

13.31 Knot Resolver 1.2.4 (2017-03-09)

13.31.1 Security

- Knot Resolver 1.2.0 and higher could return AD flag for insecure answer if the daemon received answer with invalid RRSIG several times in a row.

13.31.2 Improvements

- modules/policy: allow QTRACE policy to be chained with other policies
- hints.add_hosts(path): a new property

- module: document the API and simplify the code
- policy.MIRROR: support IPv6 link-local addresses
- policy.FORWARD: support IPv6 link-local addresses
- add net.outgoing_{v4,v6} to allow specifying address to use for connections

13.31.3 Bugfixes

- layer/iterate: some improvements in cname chain unrolling
- layer/validate: fix duplicate records in AUTHORITY section in case of WC expansion proof
- lua: do *not* truncate cache size to unsigned
- forwarding mode: correctly forward +cd flag
- fix a potential memory leak
- don't treat answers that contain DS non-existance proof as insecure
- don't store NSEC3 and their signatures in the cache
- layer/iterate: when processing delegations, check if qname is at or below new authority

13.32 Knot Resolver 1.2.3 (2017-02-23)

13.32.1 Bugfixes

- Disable storing GLUE records into the cache even in the (non-default) QUERY_PERMISSIVE mode
- iterate: skip answer RRs that don't match the query
- layer/iterate: some additional processing for referrals
- lib/resolve: zonecut fetching error was fixed

13.33 Knot Resolver 1.2.2 (2017-02-10)

13.33.1 Bugfixes:

- Fix -k argument processing to avoid out-of-bounds memory accesses
- lib/resolve: fix zonecut fetching for explicit DS queries
- hints: more NULL checks
- Fix TA bootstrapping for multiple TAs in the IANA XML file

13.33.2 Testing:

- Update tests to run tests with and without QNAME minimization

13.34 Knot Resolver 1.2.1 (2017-02-01)

13.34.1 Security:

- Under certain conditions, a cached negative answer from a CD query would be reused to construct response for non-CD queries, resulting in Insecure status instead of Bogus. Only 1.2.0 release was affected.

13.34.2 Documentation

- Update the typo in the documentation: The query trace policy is named policy.QTRACE (and not policy.TRACE)

13.34.3 Bugfixes:

- lua: make the map command check its arguments

13.35 Knot Resolver 1.2.0 (2017-01-24)

13.35.1 Security:

- In a policy.FORWARD() mode, the AD flag was being always set by mistake. It is now cleared, as the policy.FORWARD() doesn't do DNSSEC validation yet.

13.35.2 Improvements:

- The DNSSEC Validation has been refactored, fixing many resolving failures.
- Add module *version* that checks for updates and CVEs periodically.
- Support RFC7830: EDNS(0) padding in responses over TLS.
- Support CD flag on incoming requests.
- hints module: previously /etc/hosts was loaded by default, but not anymore. Users can now actually avoid loading any file.
- DNS over TLS now creates ephemeral certs.
- Configurable cache.{min,max}_tll option, with max_ttl defaulting to 6 days.
- Option to reorder RRs in the response.
- New policy.QTRACE policy to print packet contents

13.35.3 Bugfixes:

- Trust Anchor configuration is now more robust.
- Correctly answer NOTIMPL for meta-types and non-IN RR classes.
- Free TCP buffer on cancelled connection.
- Fix crash in hints module on empty hints file, and fix non-lowercase hints.

13.35.4 Miscellaneous:

- It now requires knot >= 2.3.1 to link successfully.
- The API+ABI for modules changed slightly.
- New LRU implementation.

13.36 Knot Resolver 1.1.1 (2016-08-24)

13.36.1 Bugfixes:

- Fix 0x20 randomization with retransmit
- Fix pass-through for the stub mode
- Fix the root hints IPv6 addresses
- Fix dst addr for retries over TCP

13.36.2 Improvements:

- Track RTT of all tried servers for faster retransmit
- DAF: Allow forwarding to custom port
- systemd: Read EnvironmentFile and user $KRESD_ARGS
- systemd: Update systemd units to be named after daemon

13.37 Knot Resolver 1.1.0 (2016-08-12)

13.37.1 Improvements:

- RFC7873 DNS Cookies
- RFC7858 DNS over TLS
- HTTP/2 web interface, RESTful API
- Metrics exported in Prometheus
- DNS firewall module
- Explicit CNAME target fetching in strict mode
- Query minimisation improvements
- Improved integration with systemd

13.38 Knot Resolver 1.0.0 (2016-05-30)

13.38.1 Initial release:

- The first initial release

CHAPTER 14

Building from sources

Note: Latest up-to-date packages for various distribution can be obtained from web https://knot-resolver.cz/download/.

Knot Resolver is written for UNIX-like systems using modern C standards. Beware that some 64-bit systems with LuaJIT 2.1 may be affected by a problem – Linux on x86_64 is unaffected but Linux on aarch64 is.

```
$ git clone --recursive https://gitlab.labs.nic.cz/knot/knot-resolver.git
```

14.1 Dependencies

Note: This section lists basic requirements. Individual modules might have additional build or runtime dependencies.

The following dependencies are needed to build and run Knot Resolver:

Requirement	Notes
ninja	*build only*
meson >= 0.46	*build only*[1]
C and C++ compiler	*build only*[2]
pkg-config	*build only*[3]
libknot 2.8+	Knot DNS libraries
LuaJIT 2.0+	Embedded scripting language
libuv 1.7+	Multiplatform I/O and services
lmdb	Memory-mapped database for cache
GnuTLS	TLS

There are also *optional* packages that enable specific functionality in Knot Resolver:

Optional	Needed for	Notes
lua-http	`modules/http`	HTTP/2 client/server for Lua.
lua-cqueues	`modules/graphite`	Send statistics over the Graphite protocol.
cmocka	`unit tests`	Unit testing framework.
Doxygen	`documentation`	Generating API documentation.
Sphinx and sphinx_rtd_theme	`documentation`	Building this HTML/PDF documentation.
breathe	`documentation`	Exposing Doxygen API doc to Sphinx.
libsystemd	`daemon`	Systemd watchdog support.
libprotobuf 3.0+	`modules/dnstap`	Protocol Buffers support for dnstap.
libprotobuf-c 1.0+	`modules/dnstap`	C bindings for Protobuf.
libfstrm 0.2+	`modules/dnstap`	Frame Streams data transport protocol.
luacheck	`lint-lua`	Syntax and static analysis checker for Lua.
clang-tidy	`lint-c`	Syntax and static analysis checker for C.
luacov	`check-config`	Code coverage analysis for Lua modules.

14.1.1 Packaged dependencies

Note: Some build dependencies can be found in home:CZ-NIC:knot-resolver-build.

On reasonably new systems most of the dependencies can be resolved from packages, here's an overview for several platforms.

- **Debian/Ubuntu** - Current stable doesn't have new enough Meson and libknot. Use repository above or build them yourself. Fresh list of dependencies can be found in Debian control file in our repo, search for "Build-Depends".
- **CentOS/Fedora/RHEL/openSUSE** - Fresh list of dependencies can be found in RPM spec file in our repo, search for "BuildRequires".
- **FreeBSD** - when installing from ports, all dependencies will install automatically, corresponding to the selected options.
- **Mac OS X** - the dependencies can be obtained from Homebrew formula.

14.2 Compilation

Note: Knot Resolver uses Meson Build system. Shell snippets below should be sufficient for basic usage but users unfamiliar with Meson Build might want to read introductory article Using Meson.

Following example script will:

- create new build directory named `build_dir`
- configure installation path `/tmp/kr`

[1] If `meson >= 0.46` isn't available for your distro, check backports repository or use python pip to install it.

[2] Requires `__attribute__((cleanup))` and `-MMD -MP` for dependency file generation. We test GCC and Clang, and ICC is likely to work as well.

[3] You can use variables `<dependency>_CFLAGS` and `<dependency>_LIBS` to configure dependencies manually (i.e. `libknot_CFLAGS` and `libknot_LIBS`).

- enable static build (to allow installation to non-standard path)
- build Knot Resolver
- install it into the previously configured path

```
$ meson build_dir --prefix=/tmp/kr --default-library=static
$ ninja -C build_dir
$ ninja install -C build_dir
```

At this point you can execute the newly installed binary using path `/tmp/kr/sbin/kresd`.

Note: When compiling on OS X, creating a shared library is currently not possible when using luajit package from Homebrew due to #37169.

14.2.1 Build options

It's possible to change the compilation with build options. These are useful to packagers or developers who wish to customize the daemon behaviour, run extended test suites etc. By default, these are all set to sensible values.

For complete list of build options create a build directory and run:

```
$ meson build_dir
$ meson configure build_dir
```

To customize project build options, use `-Doption=value` when creating a build directory:

```
$ meson build_dir -Ddoc=enabled
```

... or change options in an already existing build directory:

```
$ meson configure build_dir -Ddoc=enabled
```

14.2.2 Customizing compiler flags

If you'd like to use customize the build, see meson's built-in options. For hardening, see `b_pie`.

For complete control over the build flags, use `--buildtype=plain` and set `CFLAGS`, `LDFLAGS` when creating the build directory with `meson` command.

14.3 Tests

The following command runs all enabled tests. By default, only unit tests are enabled.

```
$ ninja -C build_dir
$ meson test -C build_dir
```

More comprehensive tests require you to install `kresd` into the configured prefix before running the test suite. They also have to be explicitly enabled by using either `-Dconfig_tests=enabled` for postinstall config tests, or `-Dextra_tests=enabled` for all tests, including deckard tests. Please note the latter also requires `-Dsendmmsg=disabled`.

```
$ meson configure build_dir -Dconfig_tests=enabled
$ ninja install -C build_dir
$ meson test -C build_dir
```

It's also possible to run only specific test suite or a test.

```
$ meson test -C build_dir --help
$ meson test -C build_dir --list
$ meson test -C build_dir --no-suite postinstall
$ meson test -C build_dir integration.serve_stale
```

14.4 HTML Documentation

To check for documentation dependencies and allow its installation, use `-Ddoc=enabled`. The documentation doesn't build automatically. Instead, target `doc` must be called explicitly.

```
$ meson build_dir -Ddoc=enabled
$ ninja -C build_dir doc
```

14.5 Tarball

Released tarballs are available from https://knot-resolver.cz/download/

To make a release tarball from git, use the follwing command. The

```
$ ninja -C build_dir dist
```

It's also possible to make a development snapshot tarball:

```
$ ./scripts/make-dev-archive.sh
```

14.6 Packaging

Recommended build options for packagers:

- `--buildtype=release` for default flags (optimalization, asserts, ...). For complete control over flags, use `plain` and see *Customizing compiler flags*.
- `--prefix=/usr` to customize prefix, other directories can be set in a similar fashion, see `meson setup --help`
- `-Dsystemd_files=enabled` for systemd unit files
- `-Ddoc=enabled` for offline html documentation (see *HTML Documentation*)
- `-Dinstall_kresd_conf=enabled` to install default config file
- `-Dclient=enabled` to force build of kresc
- `-Dunit_tests=enabled` to force build of unit tests

14.6.1 Systemd

It's recommended to use the upstream system unit files. If any customizations are required, drop-in files should be used, instead of patching/changing the unit files themselves.

To install systemd unit files, use the `-Dsystemd_files=enabled` build option.

To support enabling services after boot, you must also link `kresd.target` to `multi-user.target.wants`:

```
ln -s ../kresd.target /usr/lib/systemd/system/multi-user.target.wants/kresd.target
```

14.6.2 Trust anchors

If the target distro has externally managed (read-only) DNSSEC trust anchors or root hints use this:

- `-Dkeyfile_default=/usr/share/dns/root.key`
- `-Droot_hints=/usr/share/dns/root.hints`
- `-Dmanaged_ta=disabled`

In case you want to have automatically managed DNSSEC trust anchors instead, set `-Dmanaged_ta=enabled` and make sure both `keyfile_default` file and its parent directories are writable by kresd process (after package installation!).

14.7 Docker image

Visit hub.docker.com/r/cznic/knot-resolver for instructions how to run the container.

For development, it's possible to build the container directly from your git tree:

```
$ docker build -t knot-resolver .
```

CHAPTER 15

Custom HTTP services

This chapter describes how to create custom HTTP services inside Knot Resolver. Please read HTTP module basics in chapter *HTTP services* before continuing.

Each network address+protocol+port combination configured using `net.listen()` is associated with *kind* of endpoint, e.g. `doh` or `webmgmt`.

Each of these *kind* names is associated with table of HTTP endpoints, and the default table can be replaced using `http.config()` configuration call which allows your to provide your own HTTP endpoints.

Items in the table of HTTP endpoints are small tables describing a triplet - `{mime, on_serve, on_websocket}`. In order to register a new service in `webmgmt` *kind* of HTTP endpoint add the new endpoint description to respective table:

```
-- custom function to handle HTTP /health requests
local on_health = {'application/json',
function (h, stream)
        -- API call, return a JSON table
        return {state = 'up', uptime = 0}
end,
function (h, ws)
        -- Stream current status every second
        local ok = true
        while ok do
                local push = tojson('up')
                ok = ws:send(tojson({'up'}))
                require('cqueues').sleep(1)
        end
        -- Finalize the WebSocket
        ws:close()
end}

modules.load('http')
-- copy all existing webmgmt endpoints
my_mgmt_endpoints = http.configs._builtin.webmgmt.endpoints
-- add custom endpoint to the copy
```

(continues on next page)

(continued from previous page)

```
my_mgmt_endpoints['/health'] = on_health
-- use custom HTTP configuration for webmgmt
http.config({
        endpoints = my_mgmt_endpoints
}, 'webmgmt')
```

Then you can query the API endpoint, or tail the WebSocket using curl.

```
$ curl -k https://localhost:8453/health
{"state":"up","uptime":0}
$ curl -k -i -N -H "Connection: Upgrade" -H "Upgrade: websocket" -H "Host:␣
↪localhost:8453/health"  -H "Sec-Websocket-Key: nope" -H "Sec-Websocket-Version: 13"␣
↪https://localhost:8453/health
HTTP/1.1 101 Switching Protocols
upgrade: websocket
sec-websocket-accept: eg18mwU7CDRGUF1Q+EJwPM335eM=
connection: upgrade

?["up"]?["up"]?["up"]
```

Since the stream handlers are effectively coroutines, you are free to keep state and yield using cqueues library.

This is especially useful for WebSockets, as you can stream content in a simple loop instead of chains of callbacks.

Last thing you can publish from modules are *"snippets"*. Snippets are plain pieces of HTML code that are rendered at the end of the built-in webpage. The snippets can be extended with JS code to talk to already exported restful APIs and subscribe to WebSockets.

```
http.snippets['/health'] = {'Health service', '<p>UP!</p>'}
```

15.1 Custom RESTful services

A RESTful service is likely to respond differently to different type of methods and requests, there are three things that you can do in a service handler to send back results. First is to just send whatever you want to send back, it has to respect MIME type that the service declared in the endpoint definition. The response code would then be `200 OK`, any non-string responses will be packed to JSON. Alternatively, you can respond with a number corresponding to the HTTP response code or send headers and body yourself.

```
-- Our upvalue
local value = 42

-- Expose the service
local service = {'application/json',
function (h, stream)
        -- Get request method and deal with it properly
        local m = h:get(':method')
        local path = h:get(':path')
        log('[service] method %s path %s', m, path)
        -- Return table, response code will be '200 OK'
        if m == 'GET' then
                return {key = path, value = value}
        -- Save body, perform check and either respond with 505 or 200 OK
        elseif m == 'POST' then
                local data = stream:get_body_as_string()
```

(continues on next page)

(continued from previous page)

```
                if not tonumber(data) then
                        return 500, 'Not a good request'
                end
                value = tonumber(data)
        -- Unsupported method, return 405 Method not allowed
        else
                return 405, 'Cannot do that'
        end
end}
modules.load('http')
http.config({
        endpoints = { ['/service'] = service }
}, 'myservice')
-- do not forget to create socket of new kind using
-- net.listen(..., { kind = 'myservice' })
-- or configure systemd socket kresd-myservice.socket
```

In some cases you might need to send back your own headers instead of default provided by HTTP handler, you can do this, but then you have to return `false` to notify handler that it shouldn't try to generate a response.

```
local headers = require('http.headers')
function (h, stream)
        -- Send back headers
        local hsend = headers.new()
        hsend:append(':status', '200')
        hsend:append('content-type', 'binary/octet-stream')
        assert(stream:write_headers(hsend, false))
        -- Send back data
        local data = 'binary-data'
        assert(stream:write_chunk(data, true))
        -- Disable default handler action
        return false
end
```

CHAPTER 16

Knot Resolver library

16.1 Requirements

- libknot 2.0 (Knot DNS high-performance DNS library.)

16.2 For users

The library as described provides basic services for name resolution, which should cover the usage, examples are in the *resolve API* documentation.

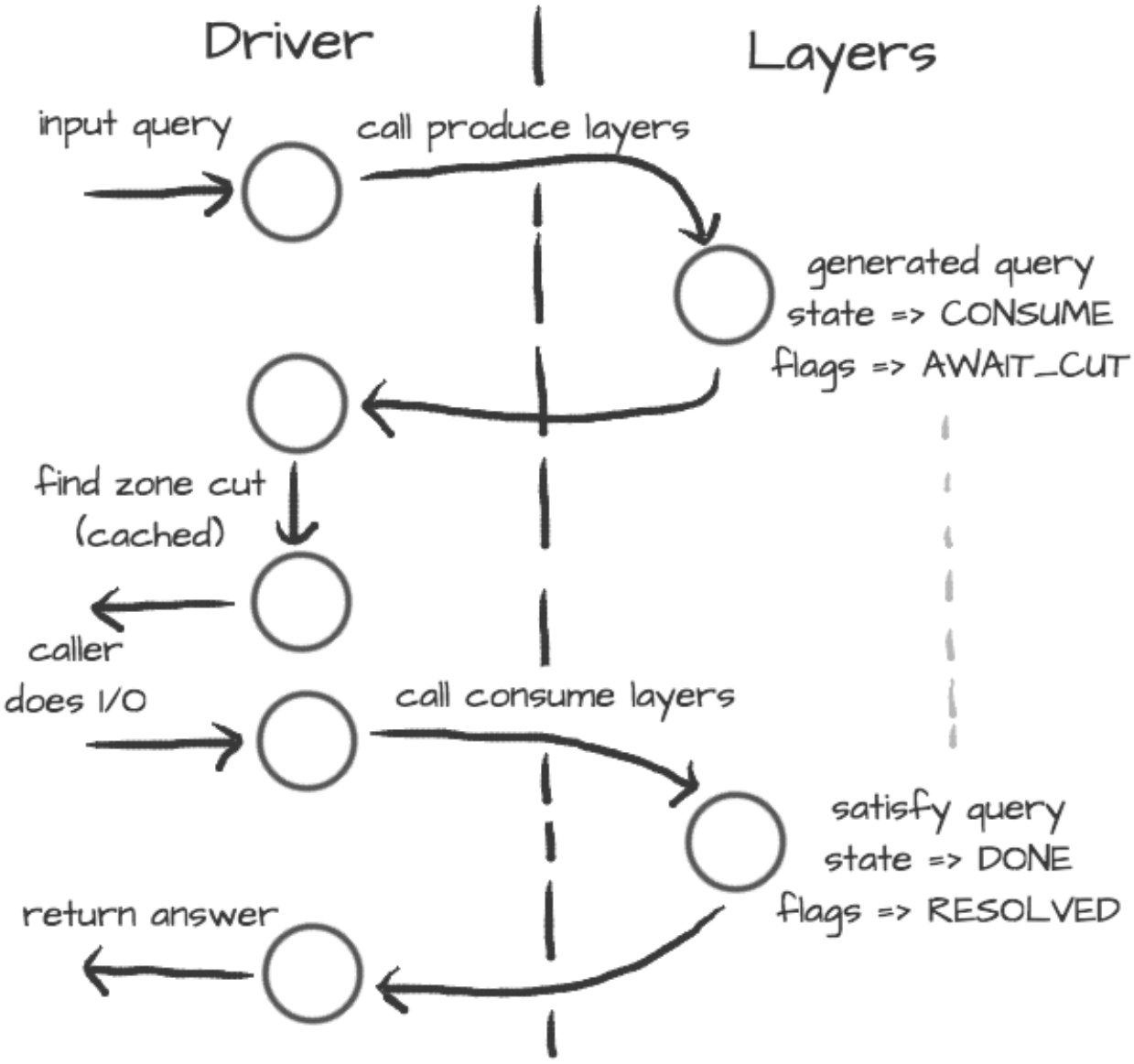
Tip: If you're migrating from `getaddrinfo()`, see *"synchronous"* API, but the library offers iterative API as well to plug it into your event loop for example.

16.3 For developers

The resolution process starts with the functions in *resolve.c*, they are responsible for:

- reacting to state machine state (i.e. calling consume layers if we have an answer ready)
- interacting with the library user (i.e. asking caller for I/O, accepting queries)
- fetching assets needed by layers (i.e. zone cut)

This is the *driver*. The driver is not meant to know *"how"* the query resolves, but rather *"when"* to execute *"what"*.



On the other side are *layers*. They are responsible for dissecting the packets and informing the driver about the results. For example, a *produce* layer generates query, a *consume* layer validates answer.

Tip: Layers are executed asynchronously by the driver. If you need some asset beforehand, you can signalize the driver using returning state or current query flags. For example, setting a flag `AWAIT_CUT` forces driver to fetch zone cut information before the packet is consumed; setting a `RESOLVED` flag makes it pop a query after the current set of layers is finished; returning `FAIL` state makes it fail current query.

Layers can also change course of resolution, for example by appending additional queries.

```
consume = function (state, req, answer)
        if answer:qtype() == kres.type.NS then
                local qry = req:push(answer:qname(), kres.type.SOA, kres.class.IN)
```

(continues on next page)

(continued from previous page)

```
                qry.flags.AWAIT_CUT = true
        end
        return state
end
```

This **doesn't** block currently processed query, and the newly created sub-request will start as soon as driver finishes processing current. In some cases you might need to issue sub-request and process it **before** continuing with the current, i.e. validator may need a DNSKEY before it can validate signatures. In this case, layers can yield and resume afterwards.

```
consume = function (state, req, answer)
        if state == kres.YIELD then
                print('continuing yielded layer')
                return kres.DONE
        else
                if answer:qtype() == kres.type.NS then
                        local qry = req:push(answer:qname(), kres.type.SOA, kres.
↪class.IN)
                        qry.flags.AWAIT_CUT = true
                        print('planned SOA query, yielding')
                        return kres.YIELD
                end
                return state
        end
end
```

The `YIELD` state is a bit special. When a layer returns it, it interrupts current walk through the layers. When the layer receives it, it means that it yielded before and now it is resumed. This is useful in a situation where you need a sub-request to determine whether current answer is valid or not.

16.4 Writing layers

Warning: FIXME: this dev-docs section is outdated! Better see comments in files instead, for now.

The resolver *library* leverages the processing API from the libknot to separate packet processing code into layers.

Note: This is only crash-course in the library internals, see the resolver *library* documentation for the complete overview of the services.

The library offers following services:

- *Cache* - MVCC cache interface for retrieving/storing resource records.
- *Resolution plan* - Query resolution plan, a list of partial queries (with hierarchy) sent in order to satisfy original query. This contains information about the queries, nameserver choice, timing information, answer and its class.
- *Nameservers* - Reputation database of nameservers, this serves as an aid for nameserver choice.

A processing layer is going to be called by the query resolution driver for each query, so you're going to work with *struct kr_request* as your per-query context. This structure contains pointers to resolution context, resolution plan and also the final answer.

```
int consume(kr_layer_t *ctx, knot_pkt_t *pkt)
{
        struct kr_request *req = ctx->req;
        struct kr_query *qry = req->current_query;
}
```

This is only passive processing of the incoming answer. If you want to change the course of resolution, say satisfy a query from a local cache before the library issues a query to the nameserver, you can use states (see the *Static hints* for example).

```
int produce(kr_layer_t *ctx, knot_pkt_t *pkt)
{
        struct kr_request *req = ctx->req;
        struct kr_query *qry = req->current_query;

        /* Query can be satisfied locally. */
        if (can_satisfy(qry)) {
                /* This flag makes the resolver move the query
                 * to the "resolved" list. */
                qry->flags.RESOLVED = true;
                return KR_STATE_DONE;
        }

        /* Pass-through. */
        return ctx->state;
}
```

It is possible to not only act during the query resolution, but also to view the complete resolution plan afterwards. This is useful for analysis-type tasks, or *"per answer"* hooks.

```
int finish(kr_layer_t *ctx)
{
        struct kr_request *req = ctx->req;
        struct kr_rplan *rplan = req->rplan;

        /* Print the query sequence with start time. */
        char qname_str[KNOT_DNAME_MAXLEN];
        struct kr_query *qry = NULL
        WALK_LIST(qry, rplan->resolved) {
                knot_dname_to_str(qname_str, qry->sname, sizeof(qname_str));
                printf("%s at %u\n", qname_str, qry->timestamp);
        }

        return ctx->state;
}
```

16.5 APIs in Lua

The APIs in Lua world try to mirror the C APIs using LuaJIT FFI, with several differences and enhancements. There is not comprehensive guide on the API yet, but you can have a look at the bindings file.

16.5.1 Elementary types and constants

- States are directly in `kres` table, e.g. `kres.YIELD`, `kres.CONSUME`, `kres.PRODUCE`, `kres.DONE`, `kres.FAIL`.
- DNS classes are in `kres.class` table, e.g. `kres.class.IN` for Internet class.
- DNS types are in `kres.type` table, e.g. `kres.type.AAAA` for AAAA type.
- DNS rcodes types are in `kres.rcode` table, e.g. `kres.rcode.NOERROR`.
- Packet sections (QUESTION, ANSWER, AUTHORITY, ADDITIONAL) are in the `kres.section` table.

16.5.2 Working with domain names

The internal API usually works with domain names in label format, you can convert between text and wire freely.

```
local dname = kres.str2dname('business.se')
local strname = kres.dname2str(dname)
```

16.5.3 Working with resource records

Resource records are stored as tables.

```
local rr = { owner = kres.str2dname('owner'),
             ttl = 0,
             class = kres.class.IN,
             type = kres.type.CNAME,
             rdata = kres.str2dname('someplace') }
print(kres.rr2str(rr))
```

RRSets in packet can be accessed using FFI, you can easily fetch single records.

```
local rrset = { ... }
local rr = rrset:get(0) -- Return first RR
print(kres.dname2str(rr:owner()))
print(rr:ttl())
print(kres.rr2str(rr))
```

16.5.4 Working with packets

Packet is the data structure that you're going to see in layers very often. They consists of a header, and four sections: QUESTION, ANSWER, AUTHORITY, ADDITIONAL. The first section is special, as it contains the query name, type, and class; the rest of the sections contain RRSets.

First you need to convert it to a type known to FFI and check basic properties. Let's start with a snippet of a *consume* layer.

```
consume = function (state, req, pkt)
        print('rcode:', pkt:rcode())
        print('query:', kres.dname2str(pkt:qname()), pkt:qclass(), pkt:qtype())
        if pkt:rcode() ~= kres.rcode.NOERROR then
                print('error response')
        end
end
```

You can enumerate records in the sections.

```
local records = pkt:section(kres.section.ANSWER)
for i = 1, #records do
        local rr = records[i]
        if rr.type == kres.type.AAAA then
                print(kres.rr2str(rr))
        end
end
```

During *produce* or *begin*, you might want to want to write to packet. Keep in mind that you have to write packet sections in sequence, e.g. you can't write to ANSWER after writing AUTHORITY, it's like stages where you can't go back.

```
pkt:rcode(kres.rcode.NXDOMAIN)
-- Clear answer and write QUESTION
pkt:recycle()
pkt:question('\7blocked', kres.class.IN, kres.type.SOA)
-- Start writing data
pkt:begin(kres.section.ANSWER)
-- Nothing in answer
pkt:begin(kres.section.AUTHORITY)
local soa = { owner = '\7blocked', ttl = 900, class = kres.class.IN, type = kres.type.
↪SOA, rdata = '...' }
pkt:put(soa.owner, soa.ttl, soa.class, soa.type, soa.rdata)
```

16.5.5 Working with requests

The request holds information about currently processed query, enabled options, cache, and other extra data. You primarily need to retrieve currently processed query.

```
consume = function (state, req, pkt)
        print(req.options)
        print(req.state)

        -- Print information about current query
        local current = req:current()
        print(kres.dname2str(current.owner))
        print(current.stype, current.sclass, current.id, current.flags)
end
```

In layers that either begin or finalize, you can walk the list of resolved queries.

```
local last = req:resolved()
print(last.stype)
```

As described in the layers, you can not only retrieve information about current query, but also push new ones or pop old ones.

```
-- Push new query
local qry = req:push(pkt:qname(), kres.type.SOA, kres.class.IN)
qry.flags.AWAIT_CUT = true

-- Pop the query, this will erase it from resolution plan
req:pop(qry)
```

16.5.6 Significant Lua API changes

Incompatible changes since 3.0.0

In the main `kres.*` lua binding, there was only change in struct knot_rrset_t:

- constructor now accepts TTL as additional parameter (defaulting to zero)
- add_rdata() doesn't accept TTL anymore (and will throw an error if passed)

In case you used knot_* functions and structures bound to lua:

- knot_dname_is_sub(a, b): knot_dname_in_bailiwick(a, b) > 0
- knot_rdata_rdlen(): knot_rdataset_at().len
- knot_rdata_data(): knot_rdataset_at().data
- knot_rdata_array_size(): offsetof(struct knot_data_t, data) + knot_rdataset_at().len
- struct knot_rdataset: field names were renamed to .count and .rdata
- some functions got inlined from headers, but you can use their kr_* clones: kr_rrsig_sig_inception(), kr_rrsig_sig_expiration(), kr_rrsig_type_covered(). Note that these functions now accept knot_rdata_t* instead of a pair knot_rdataset_t* and size_t - you can use knot_rdataset_at() for that.
- knot_rrset_add_rdata() doesn't take TTL parameter anymore
- knot_rrset_init_empty() was inlined, but in lua you can use the constructor
- knot_rrset_ttl() was inlined, but in lua you can use :ttl() method instead
- knot_pkt_qname(), _qtype(), _qclass(), _rr(), _section() were inlined, but in lua you can use methods instead, e.g. myPacket:qname()
- knot_pkt_free() takes knot_pkt_t* instead of knot_pkt_t**, but from lua you probably didn't want to use that; constructor ensures garbage collection.

16.6 API reference

16.6.1 Name resolution

The API provides an API providing a "consumer-producer"-like interface to enable user to plug it into existing event loop or I/O code.

Example usage of the iterative API:

```
// Create request and its memory pool
struct kr_request req = {
    .pool = {
        .ctx = mp_new (4096),
        .alloc = (mm_alloc_t) mp_alloc
    }
};

// Setup and provide input query
int state = kr_resolve_begin(&req, ctx, final_answer);
state = kr_resolve_consume(&req, query);

// Generate answer
while (state == KR_STATE_PRODUCE) {

    // Additional query generate, do the I/O and pass back answer
    state = kr_resolve_produce(&req, &addr, &type, query);
    while (state == KR_STATE_CONSUME) {
        int ret = sendrecv(addr, proto, query, resp);

        // If I/O fails, make "resp" empty
        state = kr_resolve_consume(&request, addr, resp);
        knot_pkt_clear(resp);
    }
    knot_pkt_clear(query);
}

// "state" is either DONE or FAIL
kr_resolve_finish(&request, state);
```

Defines

kr_request_selected(req)
Initializer for an array of *_selected.

Enums

kr_rank
RRset rank - for cache and ranked_rr_*.

The rank meaning consists of one independent flag - KR_RANK_AUTH, and the rest have meaning of values where only one can hold at any time. You can use one of the enums as a safe initial value, optionally | KR_RANK_AUTH; otherwise it's best to manipulate ranks via the kr_rank_* functions.

See also: https://tools.ietf.org/html/rfc2181#section-5.4.1 https://tools.ietf.org/html/rfc4035#section-4.3

Note The representation is complicated by restrictions on integer comparison:

- AUTH must be > than !AUTH
- AUTH INSECURE must be > than AUTH (because it attempted validation)
- !AUTH SECURE must be > than AUTH (because it's valid)

Values:

0
Did not attempt to validate.

It's assumed compulsory to validate (or prove insecure).

KR_RANK_OMIT
Do not attempt to validate.

(And don't consider it a validation failure.)

KR_RANK_TRY
Attempt to validate, but failures are non-fatal.

4
Unable to determine whether it should be secure.

KR_RANK_BOGUS
Ought to be secure but isn't.

KR_RANK_MISMATCH

KR_RANK_MISSING
Unable to obtain a good signature.

8
Proven to be insecure, i.e.

we have a chain of trust from TAs that cryptographically denies the possibility of existence of a positive chain of trust from the TAs to the record.

16
Authoritative data flag; the chain of authority was "verified".

Even if not set, only in-bailiwick stuff is acceptable, i.e. almost authoritative (example: mandatory glue and its NS RR).

32
Verified whole chain of trust from the closest TA.

Functions

bool **kr_rank_check**(uint8_t *rank*)
Check that a rank value is valid.

Meant for assertions.

static bool **kr_rank_test**(uint8_t *rank*, uint8_t *kr_flag*)
Test the presence of any flag/state in a rank, i.e.

including KR_RANK_AUTH.

static void **kr_rank_set**(uint8_t * *rank*, uint8_t *kr_flag*)
Set the rank state.

The _AUTH flag is kept as it was.

KR_EXPORT int **kr_resolve_begin**(struct *kr_request* * *request*, struct *kr_context* * *ctx*, knot_pkt_t * *answer*)
Begin name resolution.

Note Expects a request to have an initialized mempool, the "answer" packet will be kept during the resolution and will contain the final answer at the end.

Return CONSUME (expecting query)

Parameters

- `request`: request state with initialized mempool
- `ctx`: resolution context
- `answer`: allocated packet for final answer

KR_EXPORT int **kr_resolve_consume**(struct *kr_request* * *request*, const struct sockaddr * *src*, knot_pkt_t * *packet*)

Consume input packet (may be either first query or answer to query originated from *kr_resolve_produce()*)

Note If the I/O fails, provide an empty or NULL packet, this will make iterator recognize nameserver failure.

Return any state

Parameters

- `request`: request state (awaiting input)
- `src`: [in] packet source address
- `packet`: [in] input packet

KR_EXPORT int **kr_resolve_produce**(struct *kr_request* * *request*, struct sockaddr ** *dst*, int * *type*, knot_pkt_t * *packet*)

Produce either next additional query or finish.

If the CONSUME is returned then dst, type and packet will be filled with appropriate values and caller is responsible to send them and receive answer. If it returns any other state, then content of the variables is undefined.

Return any state

Parameters

- `request`: request state (in PRODUCE state)
- `dst`: [out] possible address of the next nameserver
- `type`: [out] possible used socket type (SOCK_STREAM, SOCK_DGRAM)
- `packet`: [out] packet to be filled with additional query

KR_EXPORT int **kr_resolve_checkout**(struct *kr_request* * *request*, const struct sockaddr * *src*, struct sockaddr * *dst*, int *type*, knot_pkt_t * *packet*)

Finalises the outbound query packet with the knowledge of the IP addresses.

Note The function must be called before actual sending of the request packet.

Return kr_ok() or error code

Parameters

- `request`: request state (in PRODUCE state)
- `src`: address from which the query is going to be sent
- `dst`: address of the name server
- `type`: used socket type (SOCK_STREAM, SOCK_DGRAM)
- `packet`: [in,out] query packet to be finalised

KR_EXPORT int **kr_resolve_finish**(struct *kr_request* * *request*, int *state*)

Finish resolution and commit results if the state is DONE.

Note The structures will be deinitialized, but the assigned memory pool is not going to be destroyed, as it's owned by caller.

Return DONE

Parameters

- `request`: request state
- `state`: either DONE or FAIL state (to be assigned to request->state)

KR_EXPORT KR_PURE struct *kr_rplan** **kr_resolve_plan** (struct *kr_request* * *request*)
Return resolution plan.

Return pointer to rplan

Parameters

- `request`: request state

KR_EXPORT KR_PURE knot_mm_t* **kr_resolve_pool** (struct *kr_request* * *request*)
Return memory pool associated with request.

Return mempool

Parameters

- `request`: request state

struct kr_context
#include <resolve.h> Name resolution context.

Resolution context provides basic services like cache, configuration and options.

Note This structure is persistent between name resolutions and may be shared between threads.

Public Members

struct *kr_qflags* **options**

knot_rrset_t* **opt_rr**

map_t **trust_anchors**

map_t **negative_anchors**

struct *kr_zonecut* **root_hints**

struct *kr_cache* **cache**

kr_nsrep_rtt_lru_t* **cache_rtt**

unsigned **cache_rtt_tout_retry_interval**

kr_nsrep_lru_t* **cache_rep**

module_array_t* **modules**

struct kr_cookie_ctx **cookie_ctx**

kr_cookie_lru_t* **cache_cookie**

int32_t **tls_padding**
See net.tls_padding in ../daemon/README.rst -1 is "true" (default policy), 0 is "false" (no padding)

knot_mm_t* **pool**

struct kr_request_qsource_flags

Public Members

bool *kr_request_qsource_flags*::tcp : **1**
true if the request is on TCP (or TLS); only meaningful if (dst_addr).

bool *kr_request_qsource_flags*::tls : **1**
true if the request is on TLS (or HTTPS); only meaningful if (dst_addr).

bool *kr_request_qsource_flags*::http : **1**
true if the request is on HTTP; only meaningful if (dst_addr).

struct kr_request
#include <*resolve.h*> Name resolution request.

Keeps information about current query processing between calls to processing APIs, i.e. current resolved query, resolution plan, ... Use this instead of the simple interface if you want to implement multiplexing or custom I/O.

Note All data for this request must be allocated from the given pool.

Public Members

struct *kr_context** **ctx**

knot_pkt_t* **answer**

struct *kr_query** **current_query**
Current evaluated query.

const struct sockaddr* **addr**
Address that originated the request.

Current upstream address.

NULL for internal origin.

const struct sockaddr* **dst_addr**
Address that accepted the request.

NULL for internal origin. Beware: in case of UDP on wildcard address it will be wildcard; closely related: issue #173.

const knot_pkt_t* **packet**

struct *kr_request_qsource_flags* **flags**
See definition above.

size_t **size**
query packet size

struct *kr_request*::@6 **qsource**

unsigned **rtt**
Current upstream RTT.

struct *kr_request*::@7 **upstream**
Upstream information, valid only in consume() phase.

struct *kr_qflags* **options**

int **state**

ranked_rr_array_t **answ_selected**

ranked_rr_array_t **auth_selected**

ranked_rr_array_t **add_selected**

bool **answ_validated**
internal to validator; beware of caching, etc.

bool **auth_validated**
see answ_validated ^^ ; TODO

uint8_t **rank**
Overall rank for the request.

Values from kr_rank, currently just KR_RANK_SECURE and _INITIAL. Only read this in finish phase and after validator, please. Meaning of _SECURE: all RRs in answer+authority are _SECURE, including any negative results implied (NXDOMAIN, NODATA).

struct *kr_rplan* **rplan**

trace_log_f **trace_log**
Logging tracepoint.

trace_callback_f **trace_finish**
Request finish tracepoint.

int **vars_ref**
Reference to per-request variable table.

LUA_NOREF if not set.

knot_mm_t **pool**

unsigned int **uid**

Typedefs

typedef int32_t **(* kr_stale_cb)** (int32_t *ttl*, const knot_dname_t **owner*, uint16_t *type*, const struct *kr_query* **qry*)
Callback for serve-stale decisions.

Return the adjusted TTL (typically 1) or < 0.

Parameters

- `ttl`: the expired TTL (i.e. it's < 0)

Functions

KR_EXPORT void **kr_qflags_set** (struct *kr_qflags* * *fl1*, struct *kr_qflags* *fl2*)
Combine flags together.

This means set union for simple flags.

KR_EXPORT void **kr_qflags_clear** (struct *kr_qflags* * *fl1*, struct *kr_qflags* *fl2*)
Remove flags.

This means set-theoretic difference.

KR_EXPORT int **kr_rplan_init** (struct *kr_rplan* * *rplan*, struct *kr_request* * *request*, knot_mm_t * *pool*)
Initialize resolution plan (empty).

Parameters

- `rplan`: plan instance
- `request`: resolution request
- `pool`: ephemeral memory pool for whole resolution

KR_EXPORT void **kr_rplan_deinit** (struct *kr_rplan* * *rplan*)
Deinitialize resolution plan, aborting any uncommited transactions.

Parameters

- `rplan`: plan instance

KR_EXPORT KR_PURE bool **kr_rplan_empty** (struct *kr_rplan* * *rplan*)
Return true if the resolution plan is empty (i.e.

finished or initialized)

Return true or false

Parameters

- `rplan`: plan instance

KR_EXPORT struct *kr_query** **kr_rplan_push_empty** (struct *kr_rplan* * *rplan*, struct *kr_query* * *parent*)
Push empty query to the top of the resolution plan.

Note This query serves as a cookie query only.

Return query instance or NULL

Parameters

- `rplan`: plan instance
- `parent`: query parent (or NULL)

KR_EXPORT struct *kr_query** **kr_rplan_push** (struct *kr_rplan* * *rplan*, struct *kr_query* * *parent*, const knot_dname_t * *name*, uint16_t *cls*, uint16_t *type*)
Push a query to the top of the resolution plan.

Note This means that this query takes precedence before all pending queries.

Return query instance or NULL

Parameters

- `rplan`: plan instance
- `parent`: query parent (or NULL)
- `name`: resolved name
- `cls`: resolved class
- `type`: resolved type

KR_EXPORT int **kr_rplan_pop** (struct *kr_rplan* * *rplan*, struct *kr_query* * *qry*)
Pop existing query from the resolution plan.

Note Popped queries are not discarded, but moved to the resolved list.

Return 0 or an error

Parameters

- `rplan`: plan instance
- `qry`: resolved query

KR_EXPORT KR_PURE bool **kr_rplan_satisfies**(struct *kr_query* * *closure*, const knot_dname_t * *name*, uint16_t *cls*, uint16_t *type*)
Return true if resolution chain satisfies given query.

KR_EXPORT KR_PURE struct *kr_query** **kr_rplan_resolved**(struct *kr_rplan* * *rplan*)
Return last resolved query.

KR_EXPORT KR_PURE struct *kr_query** **kr_rplan_last**(struct *kr_rplan* * *rplan*)
Return last query (either currently being solved or last resolved).

This is necessary to retrieve the last query in case of resolution failures (e.g. time limit reached).

KR_EXPORT KR_PURE struct *kr_query** **kr_rplan_find_resolved**(struct *kr_rplan* * *rplan*, struct *kr_query* * *parent*, const knot_dname_t * *name*, uint16_t *cls*, uint16_t *type*)
Check if a given query already resolved.

Return query instance or NULL

Parameters

- `rplan`: plan instance
- `parent`: query parent (or NULL)
- `name`: resolved name
- `cls`: resolved class
- `type`: resolved type

struct kr_qflags
#include <*rplan.h*> Query flags.

Public Members

bool *kr_qflags*::NO_MINIMIZE : **1**
Don't minimize QNAME.

bool *kr_qflags*::NO_THROTTLE : **1**
No query/slow NS throttling.

bool *kr_qflags*::NO_IPV6 : **1**
Disable IPv6.

bool *kr_qflags*::NO_IPV4 : **1**
Disable IPv4.

bool *kr_qflags*::TCP : **1**
Use TCP for this query.

bool *kr_qflags*::RESOLVED : 1
Query is resolved.

Note that *kr_query* gets RESOLVED before following a CNAME chain; see .CNAME.

bool *kr_qflags*::AWAIT_IPV4 : 1
Query is waiting for A address.

bool *kr_qflags*::AWAIT_IPV6 : 1
Query is waiting for AAAA address.

bool *kr_qflags*::AWAIT_CUT : 1
Query is waiting for zone cut lookup.

bool *kr_qflags*::SAFEMODE : 1
Don't use fancy stuff (EDNS, 0x20, ...)

bool *kr_qflags*::CACHED : 1
Query response is cached.

bool *kr_qflags*::NO_CACHE : 1
No cache for lookup; exception: finding NSs and subqueries.

bool *kr_qflags*::EXPIRING : 1
Query response is cached, but expiring.

bool *kr_qflags*::ALLOW_LOCAL : 1
Allow queries to local or private address ranges.

bool *kr_qflags*::DNSSEC_WANT : 1
Want DNSSEC secured answer; exception: +cd, i.e.

knot_wire_set_cd(request->answer->wire).

bool *kr_qflags*::DNSSEC_BOGUS : 1
Query response is DNSSEC bogus.

bool *kr_qflags*::DNSSEC_INSECURE : 1
Query response is DNSSEC insecure.

bool *kr_qflags*::DNSSEC_CD : 1
Instruction to set CD bit in request.

bool *kr_qflags*::STUB : 1
Stub resolution, accept received answer as solved.

bool *kr_qflags*::ALWAYS_CUT : 1
Always recover zone cut (even if cached).

bool *kr_qflags*::DNSSEC_WEXPAND : 1
Query response has wildcard expansion.

bool *kr_qflags*::PERMISSIVE : 1
Permissive resolver mode.

bool *kr_qflags*::STRICT : 1
Strict resolver mode.

bool *kr_qflags*::BADCOOKIE_AGAIN : 1
Query again because bad cookie returned.

bool *kr_qflags*::CNAME : 1
Query response contains CNAME in answer section.

bool *kr_qflags*::REORDER_RR : **1**
Reorder cached RRs.

bool *kr_qflags*::TRACE : **1**
Also log answers if verbose.

bool *kr_qflags*::NO_0X20 : **1**
Disable query case randomization .

bool *kr_qflags*::DNSSEC_NODS : **1**
DS non-existance is proven.

bool *kr_qflags*::DNSSEC_OPTOUT : **1**
Closest encloser proof has optout.

bool *kr_qflags*::NONAUTH : **1**
Non-authoritative in-bailiwick records are enough.

TODO: utilize this also outside cache.

bool *kr_qflags*::FORWARD : **1**
Forward all queries to upstream; validate answers.

bool *kr_qflags*::DNS64_MARK : **1**
Internal mark for dns64 module.

bool *kr_qflags*::CACHE_TRIED : **1**
Internal to cache module.

bool *kr_qflags*::NO_NS_FOUND : **1**
No valid NS found during last PRODUCE stage.

bool *kr_qflags*::PKT_IS_SANE : **1**
Set by iterator in consume phase to indicate whether some basic aspects of the packet are OK, e.g.

QNAME.

struct kr_query
#include <rplan.h> Single query representation.

Public Members

struct *kr_query** **parent**

knot_dname_t* **sname**
The name to resolve - lower-cased, uncompressed.

uint16_t **stype**

uint16_t **sclass**

uint16_t **id**

uint16_t **reorder**
Seed to reorder (cached) RRs in answer or zero.

struct *kr_qflags* flags **forward_flags**

uint32_t **secret**

uint32_t **uid**
Query iteration number, unique within the *kr_rplan*.

uint64_t **creation_time_mono**

uint64_t **timestamp_mono**
: Time of query created or time of query to upstream resolver (milliseconds).

struct timeval **timestamp**
: Real time for TTL+DNSSEC checks (.tv_sec only).

struct *kr_zonecut* **zone_cut**

struct *kr_layer_pickle** **deferred**

int8_t **cname_depth**
: Current xNAME depth, set by iterator.

  0 = uninitialized, 1 = no CNAME, . . . See also KR_CNAME_CHAIN_LIMIT.

struct *kr_query** **cname_parent**
: Pointer to the query that originated this one because of following a CNAME (or NULL).

struct *kr_request** **request**
: Parent resolution request.

kr_stale_cb **stale_cb**
: See the type.

struct *kr_nsrep* **ns**

struct kr_rplan
: *#include <rplan.h>* Query resolution plan structure.

The structure most importantly holds the original query, answer and the list of pending queries required to resolve the original query. It also keeps a notion of current zone cut.

Public Members

kr_qarray_t **pending**
: List of pending queries.

  Beware: order is significant ATM, as the last is the next one to solve, and they may be inter-dependent.

kr_qarray_t **resolved**
: List of resolved queries.

struct *kr_request** **request**
: Parent resolution request.

knot_mm_t* **pool**
: Temporary memory pool.

uint32_t **next_uid**
: Next value for *kr_query::uid* (incremental).

16.6.2 Cache

Functions

int **cache_peek** (*kr_layer_t* * *ctx*, knot_pkt_t * *pkt*)

int **cache_stash** (*kr_layer_t* * *ctx*, knot_pkt_t * *pkt*)

KR_EXPORT int **kr_cache_open**(struct *kr_cache* * *cache*, const struct kr_cdb_api * *api*, struct kr_cdb_opts * *opts*, knot_mm_t * *mm*)
Open/create cache with provided storage options.

Return 0 or an error code

Parameters

- `cache`: cache structure to be initialized
- `api`: storage engine API
- `opts`: storage-specific options (may be NULL for default)
- `mm`: memory context.

KR_EXPORT void **kr_cache_close**(struct *kr_cache* * *cache*)
Close persistent cache.

Note This doesn't clear the data, just closes the connection to the database.

Parameters

- `cache`: structure

KR_EXPORT int **kr_cache_commit**(struct *kr_cache* * *cache*)
Run after a row of operations to release transaction/lock if needed.

static bool **kr_cache_is_open**(struct *kr_cache* * *cache*)
Return true if cache is open and enabled.

static void **kr_cache_make_checkpoint**(struct *kr_cache* * *cache*)
(Re)set the time pair to the current values.

KR_EXPORT int **kr_cache_insert_rr**(struct *kr_cache* * *cache*, const knot_rrset_t * *rr*, const knot_rrset_t * *rrsig*, uint8_t *rank*, *uint32_t* *timestamp*)
Insert RRSet into cache, replacing any existing data.

Return 0 or an errcode

Parameters

- `cache`: cache structure
- `rr`: inserted RRSet
- `rrsig`: RRSIG for inserted RRSet (optional)
- `rank`: rank of the data
- `timestamp`: current time

KR_EXPORT int **kr_cache_clear**(struct *kr_cache* * *cache*)
Clear all items from the cache.

Return 0 or an errcode

Parameters

- `cache`: cache structure

KR_EXPORT int **kr_cache_peek_exact**(struct *kr_cache* * *cache*, const knot_dname_t * *name*, uint16_t *type*, struct *kr_cache_p* * *peek*)

KR_EXPORT int32_t **kr_cache_ttl**(const struct *kr_cache_p* * *peek*, const struct *kr_query* * *qry*, const knot_dname_t * *name*, uint16_t *type*)

KR_EXPORT int **kr_cache_materialize**(knot_rdataset_t * *dst*, const struct *kr_cache_p* * *ref*, knot_mm_t * *pool*)

KR_EXPORT int **kr_cache_remove**(struct *kr_cache* * *cache*, const knot_dname_t * *name*, uint16_t *type*)

Remove an entry from cache.

Return number of deleted records, or negative error code

Note only "exact hits" are considered ATM, and some other information may be removed alongside.

Parameters

- `cache`: cache structure
- `name`: dname
- `type`: rr type

KR_EXPORT int **kr_cache_match**(struct *kr_cache* * *cache*, const knot_dname_t * *name*, bool *exact_name*, knot_db_val_t *keyval[][2]*, int *maxcount*)

Get keys matching a dname lf prefix.

Return result count or an errcode

Note the cache keys are matched by prefix, i.e. it very much depends on their structure; CACHE_KEY_DEF.

Parameters

- `cache`: cache structure
- `name`: dname
- `exact_name`: whether to only consider exact name matches
- `keyval`: matched key-value pairs
- `maxcount`: limit on the number of returned key-value pairs

KR_EXPORT int **kr_cache_remove_subtree**(struct *kr_cache* * *cache*, const knot_dname_t * *name*, bool *exact_name*, int *maxcount*)

Remove a subtree in cache.

It's like _match but removing them instead of returning.

Return number of deleted entries or an errcode

KR_EXPORT int **kr_cache_closest_apex**(struct *kr_cache* * *cache*, const knot_dname_t * *name*, bool *is_DS*, knot_dname_t ** *apex*)

Find the closest cached zone apex for a name (in cache).

Return the number of labels to remove from the name, or negative error code

Note timestamp is found by a syscall, and stale-serving is not considered

Parameters

- `is_DS`: start searching one name higher

KR_EXPORT int **kr_unpack_cache_key**(knot_db_val_t *key*, knot_dname_t * *buf*, uint16_t * *type*)

Unpack dname and type from db key.

Return length of dname or an errcode

Note only "exact hits" are considered ATM, moreover xNAME records are "hidden" as NS. (see comments in struct entry_h)

Parameters

- `key`: db key representation
- `buf`: output buffer of domain name in dname format
- `type`: output for type

Variables

const size_t PKT_SIZE_NOWIRE = **-1**
When knot_pkt is passed from cache without ->wire, this is the ->size.

KR_EXPORT const char* **kr_cache_emergency_file_to_remove**
Path to cache file to remove on critical out-of-space error.

(do NOT modify it)

struct kr_cache
#include <api.h> Cache structure, keeps API, instance and metadata.

Public Members

knot_db_t* **db**
Storage instance.

const struct kr_cdb_api* **api**
Storage engine.

struct kr_cdb_stats **stats**

uint32_t **ttl_min**

uint32_t **ttl_max**
TTL limits.

struct timeval **checkpoint_walltime**
Wall time on the last check-point.

uint64_t **checkpoint_monotime**
Monotonic milliseconds on the last check-point.

struct kr_cache_p

Public Members

uint32_t **time**
The time of inception.

uint32_t **ttl**
TTL at inception moment.

Assuming it fits into int32_t ATM.

uint8_t **rank**
See enum kr_rank.

void* **raw_data**

void * **raw_bound**

struct *kr_cache_p*::@0 *kr_cache_p*::@**1**

16.6.3 Nameservers

Defines

KR_NS_DEAD
See kr_nsrep_update_rtt()

KR_NS_FWD_DEAD

KR_NS_TIMEOUT_RETRY_INTERVAL
If once NS was marked as "timeouted", it won't participate in NS elections at least KR_NS_TIMEOUT_RETRY_INTERVAL milliseconds (now: one second).

KR_NSREP_MAXADDR

Typedefs

typedef struct *kr_nsrep_rtt_lru_entry* **kr_nsrep_rtt_lru_entry_t**

Enums

kr_ns_score
NS RTT score (special values).

Note RTT is measured in milliseconds.

Values:

KR_CONN_RTT_MAX

2

100

10

kr_ns_rep
NS QoS flags.

Values:

0
NS has no IPv4.

1
NS has no IPv6.

2
NS has no EDNS support.

kr_ns_update_mode
NS RTT update modes.

First update is always KR_NS_RESET unless KR_NS_UPDATE_NORESET mode had choosen.

Values:

0
Update as smooth over last two measurements.

KR_NS_UPDATE_NORESET
Same as KR_NS_UPDATE, but disable fallback to KR_NS_RESET on newly added entries.

Zero is used as initial value.

KR_NS_RESET
Set to given value.

KR_NS_ADD
Increment current value.

KR_NS_MAX
Set to maximum of current/proposed value.

Functions

typedef **lru_t** (*kr_nsrep_rtt_lru_entry_t*)
NS QoS tracking.

typedef **lru_t** (unsigned)
NS reputation tracking.

KR_EXPORT int **kr_nsrep_set** (struct *kr_query* * *qry*, size_t *index*, const struct sockaddr * *sock*)
Set given NS address.

(Very low-level access to the list.)

Return 0 or an error code, in particular kr_error(ENOENT) for net.ipvX

Parameters

- `qry`: updated query
- `index`: index of the updated target
- `sock`: socket address to use (sockaddr_in or sockaddr_in6 or NULL)

KR_EXPORT int **kr_nsrep_elect** (struct *kr_query* * *qry*, struct *kr_context* * *ctx*)
Elect best nameserver/address pair from the nsset.

Return 0 or an error code

Parameters

- `qry`: updated query
- `ctx`: resolution context

KR_EXPORT int **kr_nsrep_elect_addr** (struct *kr_query* * *qry*, struct *kr_context* * *ctx*)
Elect best nameserver/address pair from the nsset.

Return 0 or an error code

Parameters

- `qry`: updated query
- `ctx`: resolution context

KR_EXPORT int **kr_nsrep_update_rtt** (struct *kr_nsrep* * *ns*, const struct sockaddr * *addr*, unsigned *score*, kr_nsrep_rtt_lru_t * *cache*, int *umode*)

Update NS address RTT information.

In KR_NS_UPDATE mode reputation is smoothed over last N measurements.

Return 0 on success, error code on failure

Parameters

- `ns`: updated NS representation
- `addr`: chosen address (NULL for first)
- `score`: new score (i.e. RTT), see enum kr_ns_score
- `cache`: RTT LRU cache
- `umode`: update mode (KR_NS_UPDATE or KR_NS_RESET or KR_NS_ADD)

KR_EXPORT int **kr_nsrep_update_rep** (struct *kr_nsrep* * *ns*, unsigned *reputation*, kr_nsrep_lru_t * *cache*)

Update NSSET reputation information.

Return 0 on success, error code on failure

Parameters

- `ns`: updated NS representation
- `reputation`: combined reputation flags, see enum kr_ns_rep
- `cache`: LRU cache

int **kr_nsrep_copy_set** (struct *kr_nsrep* * *dst*, const struct *kr_nsrep* * *src*)

Copy NSSET reputation information and resets score.

Return 0 on success, error code on failure

Parameters

- `dst`: updated NS representation
- `src`: source NS representation

KR_EXPORT int **kr_nsrep_sort** (struct *kr_nsrep* * *ns*, struct *kr_context* * *ctx*)

Sort addresses in the query nsrep list by cached RTT.

if RTT is greater then KR_NS_TIMEOUT, address will placed at the beginning of the nsrep list once in cache.ns_tout() milliseconds. Otherwise it will be sorted as if it has cached RTT equal to KR_NS_MAX_SCORE + 1.

Return 0 or an error code

Note ns reputation is zeroed and score is set to KR_NS_MAX_SCORE + 1.

Parameters

- `ns`: updated *kr_nsrep*
- `ctx`: name resolution context.

struct kr_nsrep_rtt_lru_entry

Public Members

unsigned **score**

uint64_t **tout_timestamp**

struct kr_nsrep

#include <nsrep.h> Name server representation.

Contains extra information about the name server, e.g. score or other metadata.

Public Members

unsigned **score**
NS score.

unsigned **reputation**
NS reputation.

const knot_dname_t* **name**
NS name.

struct *kr_context** **ctx**
Resolution context.

union inaddr kr_nsrep::addr[KR_NSREP_MAXADDR]
NS address(es)

Functions

KR_EXPORT int **kr_zonecut_init**(struct *kr_zonecut* * *cut*, const knot_dname_t * *name*, knot_mm_t * *pool*)
Populate root zone cut with SBELT.

Return 0 or error code

Parameters

- `cut`: zone cut
- `name`:
- `pool`:

KR_EXPORT void **kr_zonecut_deinit**(struct *kr_zonecut* * *cut*)
Clear the structure and free the address set.

Parameters

- `cut`: zone cut

KR_EXPORT void **kr_zonecut_move**(struct *kr_zonecut* * *to*, const struct *kr_zonecut* * *from*)
Move a zonecut, transferring ownership of any pointed-to memory.

Parameters

- `to`: the target - it gets deinit-ed
- `from`: the source - not modified, but shouldn't be used afterward

KR_EXPORT void **kr_zonecut_set** (struct *kr_zonecut* * *cut*, const knot_dname_t * *name*)
Reset zone cut to given name and clear address list.

Note This clears the address list even if the name doesn't change. TA and DNSKEY don't change.

Parameters

- `cut`: zone cut to be set
- `name`: new zone cut name

KR_EXPORT int **kr_zonecut_copy** (struct *kr_zonecut* * *dst*, const struct *kr_zonecut* * *src*)
Copy zone cut, including all data.

Does not copy keys and trust anchor.

Return 0 or an error code; If it fails with kr_error(ENOMEM), it may be in a half-filled state, but it's safe to deinit...

Note addresses for names in `src` get replaced and others are left as they were.

Parameters

- `dst`: destination zone cut
- `src`: source zone cut

KR_EXPORT int **kr_zonecut_copy_trust** (struct *kr_zonecut* * *dst*, const struct *kr_zonecut* * *src*)
Copy zone trust anchor and keys.

Return 0 or an error code

Parameters

- `dst`: destination zone cut
- `src`: source zone cut

KR_EXPORT int **kr_zonecut_add** (struct *kr_zonecut* * *cut*, const knot_dname_t * *ns*, const void * *data*, int *len*)
Add address record to the zone cut.

The record will be merged with existing data, it may be either A/AAAA type.

Return 0 or error code

Parameters

- `cut`: zone cut to be populated
- `ns`: nameserver name
- `data`: typically knot_rdata_t::data
- `len`: typically knot_rdata_t::len

KR_EXPORT int **kr_zonecut_del** (struct *kr_zonecut* * *cut*, const knot_dname_t * *ns*, const void * *data*, int *len*)
Delete nameserver/address pair from the zone cut.

Return 0 or error code

Parameters

- `cut`:
- `ns`: name server name
- `data`: typically knot_rdata_t::data
- `len`: typically knot_rdata_t::len

KR_EXPORT int **`kr_zonecut_del_all`**(struct *kr_zonecut* * *cut*, const knot_dname_t * *ns*)
Delete all addresses associated with the given name.

Return 0 or error code

Parameters

- `cut`:
- `ns`: name server name

KR_EXPORT KR_PURE pack_t* **`kr_zonecut_find`**(struct *kr_zonecut* * *cut*, const knot_dname_t * *ns*)
Find nameserver address list in the zone cut.

Note This can be used for membership test, a non-null pack is returned if the nameserver name exists.

Return pack of addresses or NULL

Parameters

- `cut`:
- `ns`: name server name

KR_EXPORT int **`kr_zonecut_set_sbelt`**(struct *kr_context* * *ctx*, struct *kr_zonecut* * *cut*)
Populate zone cut with a root zone using SBELT :rfc:`1034`

Return 0 or error code

Parameters

- `ctx`: resolution context (to fetch root hints)
- `cut`: zone cut to be populated

KR_EXPORT int **`kr_zonecut_find_cached`**(struct *kr_context* * *ctx*, struct *kr_zonecut* * *cut*, const knot_dname_t * *name*, const struct *kr_query* * *qry*, bool *restrict *secured*)
Populate zone cut address set from cache.

Return 0 or error code (ENOENT if it doesn't find anything)

Parameters

- `ctx`: resolution context (to fetch data from LRU caches)
- `cut`: zone cut to be populated
- `name`: QNAME to start finding zone cut for
- `qry`: query for timestamp and stale-serving decisions
- `secured`: set to true if want secured zone cut, will return false if it is provably insecure

KR_EXPORT bool **kr_zonecut_is_empty**(struct *kr_zonecut* * *cut*)
Check if any address is present in the zone cut.

Return true/false

Parameters

- `cut`: zone cut to check

struct kr_zonecut
#include <*zonecut.h*> Current zone cut representation.

Public Members

knot_dname_t* **name**
Zone cut name.

knot_rrset_t* **key**
Zone cut DNSKEY.

knot_rrset_t* **trust_anchor**
Current trust anchor.

struct *kr_zonecut** **parent**
Parent zone cut.

*trie_t** **nsset**
Map of nameserver => address_set (pack_t).

knot_mm_t* **pool**
Memory pool.

16.6.4 Modules

Module API definition and functions for (un)loading modules.

Defines

KR_MODULE_EXPORT(module)
Export module API version (place this at the end of your module).

Parameters

- `module`: module name (e.g. policy)

KR_MODULE_API

Typedefs

typedef uint32_t() *module_api_cb(void*)

typedef char*() kr_prop_cb(void *env, struct kr_module *self, const char *input)
Module property callback.

Input and output is passed via a JSON encoded in a string.

Return a free-form JSON output (malloc-ated)

Note see modules_create_table_for_c() implementation for details about the input/output conversion.

Parameters

- `env`: pointer to the lua engine, i.e. struct engine *env (TODO: explicit type)
- `input`: parameter (NULL if missing/nil on lua level)

typedef int **(* kr_module_init_cb)** (struct *kr_module* *)

Functions

KR_EXPORT int **kr_module_load** (struct *kr_module* * *module*, const char * *name*, const char * *path*)
Load a C module instance into memory.

And call its init().

Return 0 or an error

Parameters

- `module`: module structure. Will be overwritten except for ->data on success.
- `name`: module name
- `path`: module search path

KR_EXPORT void **kr_module_unload** (struct *kr_module* * *module*)
Unload module instance.

Note currently used even for lua modules

Parameters

- `module`: module structure

KR_EXPORT *kr_module_init_cb* **kr_module_get_embedded** (const char * *name*)
Get embedded module's init function by name (or NULL).

struct kr_module
#include <module.h> Module representation.

The five symbols (init, ...) may be defined by the module as name_init(), etc; all are optional and missing symbols are represented as NULLs;

Public Members

char* **name**

int **init)** (struct *kr_module* *self*)
Constructor.

Called after loading the module.

Return error code. Lua modules: not populated, called via lua directly.

int **deinit)** (struct *kr_module* *self*)
Destructor.

Called before unloading the module.

Return error code.

int**config)** (struct *kr_module* *self, const char *input)
Configure with encoded JSON (NULL if missing).

Return error code. Lua modules: not used and not useful from C. When called from lua, input is JSON, like for kr_prop_cb.

const *kr_layer_api_t** **layer**
Packet processing API specs.

May be NULL. See docs on that type. Owned by the module code.

const struct *kr_prop** **props**
List of properties.

May be NULL. Terminated by { NULL, NULL, NULL }. Lua modules: not used and not useful.

void* **lib**
dlopen() handle; RTLD_DEFAULT for embedded modules; NULL for lua modules.

void* **data**
Custom data context.

struct kr_prop
#include <*module.h*> Module property (named callable).

Public Members

kr_prop_cb* **cb**

const char* **name**

const char* **info**

Defines

QRVERBOSE (_query, _cls, ...)
Print a debug message related to resolution.

Parameters

- `_query`: associated *kr_query*, may be NULL
- `_cls`: identifying string, typically of length exactly four (padded)
- `...`: printf-compatible list of parameters

Typedefs

typedef struct *kr_layer* **kr_layer_t**
Packet processing context.

typedef struct *kr_layer_api* **kr_layer_api_t**

Enums

kr_layer_state
Layer processing states.

Only one value at a time (but see TODO).

Each state represents the state machine transition, and determines readiness for the next action. See struct *kr_layer_api* for the actions.

TODO: the cookie module sometimes sets (_FAIL | _DONE) on purpose (!)

Values:

0
Consume data.

1
Produce data.

2
Finished successfully or a special case: in CONSUME phase this can be used (by iterator) to do a transition to PRODUCE phase again, in which case the packet wasn't accepted for some reason.

3
Error.

4
Paused, waiting for a sub-query.

Functions

static bool **kr_state_consistent** (enum *kr_layer_state s*)
Check that a kr_layer_state makes sense.

We're not very strict ATM.

struct kr_layer
#include <*layer.h*> Packet processing context.

Public Members

int **state**
The current state; bitmap of enum kr_layer_state.

struct *kr_request** **req**
The corresponding request.

const struct *kr_layer_api** **api**

knot_pkt_t* **pkt**
In glue for lua *kr_layer_api* it's used to pass the parameter.

struct sockaddr* **dst**
In glue for checkout layer it's used to pass the parameter.

bool **is_stream**
In glue for checkout layer it's used to pass the parameter.

struct kr_layer_api

#include <*layer.h*> Packet processing module API.

All functions return the new kr_layer_state.

Lua modules are allowed to return nil/nothing, meaning the state shall not change.

Public Members

int**begin)** (*kr_layer_t* *ctx*)
Start of processing the DNS request.

int**reset)** (*kr_layer_t* *ctx*)

int**finish)** (*kr_layer_t* *ctx*)
Paired to begin, called both on successes and failures.

int**consume)** (*kr_layer_t* *ctx*, knot_pkt_t *pkt*)
Process an answer from upstream or from cache.

Lua API: call is omitted iff (state & KR_STATE_FAIL).

int**produce)** (*kr_layer_t* *ctx*, knot_pkt_t *pkt*)
Produce either an answer to the request or a query for upstream (or fail).

Lua API: call is omitted iff (state & KR_STATE_FAIL).

int**checkout)** (*kr_layer_t* *ctx*, knot_pkt_t *packet*, struct sockaddr *dst*, int *type*)
Finalises the outbound query packet with the knowledge of the IP addresses.

The checkout layer doesn't persist the state, so canceled subrequests don't affect the resolution or rest of the processing. Lua API: call is omitted iff (state & KR_STATE_FAIL).

int**answer_finalize)** (*kr_layer_t* *ctx*)
Finalises the answer.

Last chance to affect what will get into the answer, including EDNS.

void* **data**
The C module can store anything in here.

int kr_layer_api::cb_slots[]
Internal to .

/daemon/ffimodule.c.

struct kr_layer_pickle

#include <*layer.h*> Pickled layer state (api, input, state).

Public Members

struct *kr_layer_pickle** **next**

const struct *kr_layer_api** **api**

knot_pkt_t* **pkt**

unsigned **state**

16.6.5 Utilities

Defines

kr_log_info

kr_log_error(...)

kr_log_deprecate(...)

kr_log_trace_enabled(query)
Return true if the query has request log handler installed.

VERBOSE_STATUS
Block run in verbose mode; optimized when not run.

WITH_VERBOSE(query)

kr_log_verbose

KR_DNAME_GET_STR(dname_str, dname)

KR_RRTYPE_GET_STR(rrtype_str, rrtype)

static_assert(cond, msg)

KR_RRKEY_LEN

SWAP(x, y)
Swap two places.

Note: the parameters need to be without side effects.

Typedefs

typedef void **(* trace_callback_f)**(struct *kr_request* *request)
Callback for request events.

typedef void **(* trace_log_f)**(const struct *kr_query* *query, const char *source, const char *msg)
Callback for request logging handler.

Functions

KR_EXPORT bool **kr_verbose_set**(bool *status*)
Set verbose mode.

Not available if compiled with -DNOVERBOSELOG.

KR_EXPORT **KR_PRINTF**(1)
Log a message if in verbose mode.

KR_EXPORT **KR_PRINTF**(3)
Utility for QRVERBOSE - use that instead.

Log a message through the request log handler.

Return true if the message was logged

Parameters

- `query`: current query

- `source`: message source
- `fmt`: message format

static int **strcmp_p**(const void * *p1*, const void * *p2*)
A strcmp() variant directly usable for qsort() on an array of strings.

static long **time_diff**(struct timeval * *begin*, struct timeval * *end*)
Return time difference in miliseconds.

Note based on the _BSD_SOURCE timersub() macro

static void **get_workdir**(char * *out*, size_t *len*)
Get current working directory with fallback value.

KR_EXPORT char* **kr_strcatdup**(unsigned *n*, ...)
Concatenate N strings.

KR_EXPORT void **kr_rnd_buffered**(void * *data*, unsigned int *size*)
You probably want kr_rand_* convenience functions instead.

This is a buffered version of gnutls_rnd(GNUTLS_RND_NONCE, ..)

static uint64_t **kr_rand_bytes**(unsigned int *size*)
Return a few random bytes.

static bool **kr_rand_coin**(unsigned int *nomin*, unsigned int *denomin*)
Throw a pseudo-random coin, succeeding approximately with probability nomin/denomin.

- low precision, only one byte of randomness (or none with extreme parameters)
- tip: use !kr_rand_coin() to get the complementary probability

KR_EXPORT int **kr_memreserve**(void * *baton*, char ** *mem*, size_t *elm_size*, size_t *want*, size_t * *have*)
Memory reservation routine for knot_mm_t.

KR_EXPORT int **kr_pkt_recycle**(knot_pkt_t * *pkt*)

KR_EXPORT int **kr_pkt_clear_payload**(knot_pkt_t * *pkt*)

KR_EXPORT int **kr_pkt_put**(knot_pkt_t * *pkt*, const knot_dname_t * *name*, *uint32_t* *ttl*, uint16_t *rclass*, uint16_t *rtype*, const uint8_t * *rdata*, uint16_t *rdlen*)
Construct and put record to packet.

KR_EXPORT void **kr_pkt_make_auth_header**(knot_pkt_t * *pkt*)
Set packet header suitable for authoritative answer.

(for policy module)

KR_EXPORT KR_PURE const char* **kr_inaddr**(const struct sockaddr * *addr*)
Address bytes for given family.

KR_EXPORT KR_PURE int **kr_inaddr_family**(const struct sockaddr * *addr*)
Address family.

KR_EXPORT KR_PURE int **kr_inaddr_len**(const struct sockaddr * *addr*)
Address length for given family, i.e.

sizeof(struct in*_addr).

KR_EXPORT KR_PURE int **kr_sockaddr_len**(const struct sockaddr * *addr*)
Sockaddr length for given family, i.e.

sizeof(struct sockaddr_in*).

KR_EXPORT KR_PURE int **kr_sockaddr_cmp** (const struct sockaddr * *left*, const struct sockaddr * *right*)
Compare two given sockaddr.

return 0 - addresses are equal, error code otherwise.

KR_EXPORT KR_PURE uint16_t **kr_inaddr_port** (const struct sockaddr * *addr*)
Port.

KR_EXPORT void **kr_inaddr_set_port** (struct sockaddr * *addr*, uint16_t *port*)
Set port.

KR_EXPORT int **kr_inaddr_str** (const struct sockaddr * *addr*, char * *buf*, size_t * *buflen*)
Write string representation for given address as "<addr>#<port>".

Parameters

- `[in] addr`: the raw address
- `[out] buf`: the buffer for output string
- `[inout] buflen`: the available(in) and utilized(out) length, including \0

KR_EXPORT int **kr_ntop_str** (int *family*, const void * *src*, uint16_t *port*, char * *buf*, size_t * *buflen*)
Write string representation for given address as "<addr>#<port>".

It's the same as kr_inaddr_str(), but the input address is input in native format like for inet_ntop() (4 or 16 bytes) and port must be separate parameter.

static char* **kr_straddr** (const struct sockaddr * *addr*)

KR_EXPORT KR_PURE int **kr_straddr_family** (const char * *addr*)
Return address type for string.

KR_EXPORT KR_CONST int **kr_family_len** (int *family*)
Return address length in given family (struct in*_addr).

KR_EXPORT struct sockaddr* **kr_straddr_socket** (const char * *addr*, int *port*, knot_mm_t * *pool*)
Create a sockaddr* from string+port representation.

Also accepts IPv6 link-local and AF_UNIX starting with "/" (ignoring port)

KR_EXPORT int **kr_straddr_subnet** (void * *dst*, const char * *addr*)
Parse address and return subnet length (bits).

Warning 'dst' must be at least `sizeof(struct in6_addr)` long.

KR_EXPORT int **kr_straddr_split** (const char * *instr*, char ipaddr[static *restrict(INET6_ADDRSTRLEN+1)]*, uint16_t * *port*)
Splits ip address specified as "addr@port" or "addr#port" into addr and port.

Return error code

Note Typically you follow this by kr_straddr_socket().

Note Only internet addresses are supported, i.e. no AF_UNIX sockets.

Parameters

- `instr[in]`: zero-terminated input, e.g. "192.0.2.1#12345\0"
- `ipaddr[out]`: working buffer for the port-less prefix of instr; length >= INET6_ADDRSTRLEN + 1.
- `port[out]`: written in case it's specified in instr

KR_EXPORT int **kr_straddr_join** (const char * *addr*, uint16_t *port*, char * *buf*, size_t * *buflen*)
Formats ip address and port in "addr#port" format.

and performs validation.

Note Port always formatted as five-character string with leading zeros.

Return kr_error(EINVAL) - addr or buf is NULL or buflen is 0 or addr doesn't contain a valid ip address kr_error(ENOSP) - buflen is too small

KR_EXPORT KR_PURE int **kr_bitcmp** (const char * *a*, const char * *b*, int *bits*)
Compare memory bitwise.

The semantics is "the same" as for memcmp(). The partial byte is considered with more-significant bits first, so this is e.g. suitable for comparing IP prefixes.

static uint8_t **KEY_FLAG_RANK** (const char * *key*)

static bool **KEY_COVERING_RRSIG** (const char * *key*)

KR_EXPORT int **kr_rrkey** (char * *key*, uint16_t *class*, const knot_dname_t * *owner*, uint16_t *type*, uint16_t *additional*)
Create unique null-terminated string key for RR.

Return key length if successful or an error

Parameters

- `key`: Destination buffer for key size, MUST be KR_RRKEY_LEN or larger.
- `class`: RR class.
- `owner`: RR owner name.
- `type`: RR type.
- `additional`: flags (for instance can be used for storing covered type when RR type is RRSIG).

KR_EXPORT int **kr_ranked_rrarray_add** (ranked_rr_array_t * *array*, const knot_rrset_t * *rr*, uint8_t *rank*, bool *to_wire*, *uint32_t* *qry_uid*, knot_mm_t * *pool*)
Add RRSet copy to a ranked RR array.

To convert to standard RRs inside, you need to call _finalize() afterwards, and the memory of rr->rrs.rdata has to remain until then.

KR_EXPORT int **kr_ranked_rrarray_finalize** (ranked_rr_array_t * *array*, *uint32_t* *qry_uid*, knot_mm_t * *pool*)
Finalize in_progress sets - all with matching qry_uid.

int **kr_ranked_rrarray_set_wire** (ranked_rr_array_t * *array*, bool *to_wire*, *uint32_t* *qry_uid*, bool *check_dups*, bool(**extraCheck*)(const ranked_rr_array_entry_t *))

KR_EXPORT char* **kr_pkt_text** (const knot_pkt_t * *pkt*) **Return**
Newly allocated string representation of packet. Caller has to free() returned string.

KR_PURE char* **kr_rrset_text** (const knot_rrset_t * *rr*)

static *KR_PURE* char* **kr_dname_text** (const knot_dname_t * *name*)

static *KR_CONST* char* **kr_rrtype_text** (const uint16_t *rrtype*)

KR_EXPORT char* **kr_module_call** (struct *kr_context* * *ctx*, const char * *module*, const char * *prop*, const char * *input*)
Call module property.

static uint16_t **kr_rrset_type_maysig**(const knot_rrset_t * *rr*)
Return the (covered) type of an nonempty RRset.

KR_EXPORT uint64_t **kr_now**()
The current time in monotonic milliseconds.

Note it may be outdated in case of long callbacks; see uv_now().

KR_EXPORT void **kr_uv_free_cb**(uv_handle_t * *handle*)
Call free(handle->data); it's useful e.g.

as a callback in uv_close().

int **knot_dname_lf2wire**(knot_dname_t * *dst*, uint8_t *len*, const uint8_t * *lf*)
Convert name from lookup format to wire.

See knot_dname_lf

Note len bytes are read and len+1 are written with *normal* LF, but it's also allowed that the final zero byte is omitted in LF.

Return the number of bytes written (>0) or error code (<0)

static int **kr_dname_lf**(uint8_t * *dst*, const knot_dname_t * *src*, bool *add_wildcard*)
Patched knot_dname_lf.

LF for "." has length zero instead of one, for consistency. (TODO: consistency?)

Note packet is always NULL

Parameters

- `add_wildcard`: append the wildcard label

KR_EXPORT const char* **kr_strptime_diff**(const char * *format*, const char * *time1_str*, const char * *time0_str*, double * *diff*)
Difference between two calendar times specified as strings.

Parameters

- `format[in]`: format for strptime
- `diff[out]`: result from C difftime(time1, time0)

KR_EXPORT void **kr_rrset_init**(knot_rrset_t * *rrset*, knot_dname_t * *owner*, uint16_t *type*, uint16_t *rclass*, *uint32_t* *ttl*)

KR_EXPORT uint16_t **kr_pkt_has_dnssec**(const knot_pkt_t * *pkt*)

KR_EXPORT uint16_t **kr_pkt_qclass**(const knot_pkt_t * *pkt*)

KR_EXPORT uint16_t **kr_pkt_qtype**(const knot_pkt_t * *pkt*)

KR_EXPORT *uint32_t* **kr_rrsig_sig_inception**(const knot_rdata_t * *rdata*)

KR_EXPORT *uint32_t* **kr_rrsig_sig_expiration**(const knot_rdata_t * *rdata*)

KR_EXPORT uint16_t **kr_rrsig_type_covered**(const knot_rdata_t * *rdata*)

KR_EXPORT time_t **kr_file_mtime**(const char * *fname*)

Variables

KR_EXPORT bool **kr_verbose_status**
Whether in verbose mode.

Only use this for reading.

KR_EXPORT KR_EXPORT const char* **cls**

KR_EXPORT const char const char * **fmt**

KR_EXPORT const char* **source**

const uint8_t KEY_FLAG_RRSIG = **0x02**

union inaddr
#include <utils.h> Simple storage for IPx address or AF_UNSPEC.

Public Members

struct sockaddr **ip**

struct sockaddr_in **ip4**

struct sockaddr_in6 **ip6**

Defines

KR_EXPORT

KR_CONST

KR_PURE

KR_NORETURN

KR_COLD

KR_PRINTF(n)

uint

kr_ok

kr_strerror(x)

Typedefs

typedef unsigned int **uint**

Functions

static int *KR_COLD* **kr_error**(int *x*)

16.6.6 Generics library

This small collection of "generics" was born out of frustration that I couldn't find no such thing for C. It's either bloated, has poor interface, null-checking is absent or doesn't allow custom allocation scheme. BSD-licensed (or compatible) code is allowed here, as long as it comes with a test case in *tests/test_generics.c*.

- *array* - a set of simple macros to make working with dynamic arrays easier.
- *queue* - a FIFO + LIFO queue.
- *map* - a Crit-bit tree key-value map implementation (public domain) that comes with tests.
- *set* - set abstraction implemented on top of `map` (unused now).
- *pack* - length-prefixed list of objects (i.e. array-list).
- *lru* - LRU-like hash table
- *trie* - a trie-based key-value map, taken from knot-dns

array

A set of simple macros to make working with dynamic arrays easier.

```
MIN(array_push(arr, val), other)
```

Note The C has no generics, so it is implemented mostly using macros. Be aware of that, as direct usage of the macros in the evaluating macros may lead to different expectations:

May evaluate the code twice, leading to unexpected behaviour. This is a price to pay for the absence of proper generics.

Example usage:

```
array_t(const char*) arr;
array_init(arr);

// Reserve memory in advance
if (array_reserve(arr, 2) < 0) {
    return ENOMEM;
}

// Already reserved, cannot fail
array_push(arr, "princess");
array_push(arr, "leia");

// Not reserved, may fail
if (array_push(arr, "han") < 0) {
    return ENOMEM;
}

// It does not hide what it really is
for (size_t i = 0; i < arr.len; ++i) {
    printf("%s\n", arr.at[i]);
}

// Random delete
array_del(arr, 0);
```

Defines

array_t(type)
Declare an array structure.

array_init(array)
Zero-initialize the array.

array_clear(array)
Free and zero-initialize the array (plain malloc/free).

array_clear_mm(array, free, baton)
Make the array empty and free pointed-to memory.

Mempool usage: pass mm_free and a knot_mm_t* .

array_reserve(array, n)
Reserve capacity for at least n elements.

Return 0 if success, <0 on failure

array_reserve_mm(array, n, reserve, baton)
Reserve capacity for at least n elements.

Mempool usage: pass kr_memreserve and a knot_mm_t* .

Return 0 if success, <0 on failure

array_push_mm(array, val, reserve, baton)
Push value at the end of the array, resize it if necessary.

Mempool usage: pass kr_memreserve and a knot_mm_t* .

Note May fail if the capacity is not reserved.

Return element index on success, <0 on failure

array_push(array, val)
Push value at the end of the array, resize it if necessary (plain malloc/free).

Note May fail if the capacity is not reserved.

Return element index on success, <0 on failure

array_pop(array)
Pop value from the end of the array.

array_del(array, i)
Remove value at given index.

Return 0 on success, <0 on failure

array_tail(array)
Return last element of the array.

Warning Undefined if the array is empty.

Functions

static size_t **array_next_count** (size_t *want*)
Simplified Qt containers growth strategy.

static int **array_std_reserve** (void * *baton*, char ** *mem*, size_t *elm_size*, size_t *want*, size_t * *have*)

static void **array_std_free** (void * *baton*, void * *p*)

queue

A queue, usable for FIFO and LIFO simultaneously.

Both the head and tail of the queue can be accessed and pushed to, but only the head can be popped from.

Example usage:

```
// define new queue type, and init a new queue instance
typedef queue_t(int) queue_int_t;
queue_int_t q;
queue_init(q);
// do some operations
queue_push(q, 1);
queue_push(q, 2);
queue_push(q, 3);
queue_push(q, 4);
queue_pop(q);
assert(queue_head(q) == 2);
assert(queue_tail(q) == 4);

// you may iterate
typedef queue_it_t(int) queue_it_int_t;
for (queue_it_int_t it = queue_it_begin(q); !queue_it_finished(it);
     queue_it_next(it)) {
   ++queue_it_val(it);
}
assert(queue_tail(q) == 5);

queue_push_head(q, 0);
++queue_tail(q);
assert(queue_tail(q) == 6);
// free it up
queue_deinit(q);

// you may use dynamic allocation for the type itself
queue_int_t *qm = malloc(sizeof(queue_int_t));
queue_init(*qm);
queue_deinit(*qm);
free(qm);
```

Note The implementation uses a singly linked list of blocks where each block stores an array of values (for better efficiency).

Defines

queue_t (type)
The type for queue, parametrized by value type.

queue_init(q)
Initialize a queue.

You can malloc() it the usual way.

queue_deinit(q)
De-initialize a queue: make it invalid and free any inner allocations.

queue_push(q, data)
Push data to queue's tail.

(Type-safe version; use _impl() otherwise.)

queue_push_head(q, data)
Push data to queue's head.

(Type-safe version; use _impl() otherwise.)

queue_pop(q)
Remove the element at the head.

The queue must not be empty.

queue_head(q)
Return a "reference" to the element at the head (it's an L-value).

The queue must not be empty.

queue_tail(q)
Return a "reference" to the element at the tail (it's an L-value).

The queue must not be empty.

queue_len(q)
Return the number of elements in the queue (very efficient).

queue_it_t(type)
Type for queue iterator, parametrized by value type.

It's a simple structure that owns no other resources. You may NOT use it after doing any push or pop (without _begin again).

queue_it_begin(q)
Initialize a queue iterator at the head of the queue.

If you use this in assignment (instead of initialization), you will unfortunately need to add corresponding type-cast in front. Beware: there's no type-check between queue and iterator!

queue_it_val(it)
Return a "reference" to the current element (it's an L-value) .

queue_it_finished(it)
Test if the iterator has gone past the last element.

If it has, you may not use _val or _next.

queue_it_next(it)
Advance the iterator to the next element.

map

A Crit-bit tree key-value map implementation.

Example usage:

Warning If the user provides a custom allocator, it must return addresses aligned to 2B boundary.

```
map_t map = map_make(NULL);

// Custom allocator (optional)
map.malloc = &mymalloc;
map.baton  = &mymalloc_context;

// Insert k-v pairs
int values = { 42, 53, 64 };
if (map_set(&map, "princess", &values[0]) != 0 ||
    map_set(&map, "prince", &values[1])   != 0 ||
    map_set(&map, "leia", &values[2])     != 0) {
    fail();
}

// Test membership
if (map_contains(&map, "leia")) {
    success();
}

// Prefix search
int i = 0;
int count(const char *k, void *v, void *ext) { (*(int *)ext)++; return 0; }
if (map_walk_prefixed(map, "princ", count, &i) == 0) {
    printf("%d matches\n", i);
}

// Delete
if (map_del(&map, "badkey") != 0) {
    fail(); // No such key
}

// Clear the map
map_clear(&map);
```

Defines

map_walk(map, callback, baton)

Functions

static *map_t* **map_make**(struct knot_mm * *pool*)
Creates an new empty critbit map.

Pass NULL for malloc+free.

int **map_contains**(*map_t* * *map*, const char * *str*)
Returns non-zero if map contains str.

void* **map_get**(*map_t* * *map*, const char * *str*)
Returns value if map contains str.

Note: NULL may mean two different things.

int **map_set**(*map_t* * *map*, const char * *str*, void * *val*)
Inserts str into map.

Returns 0 if new, 1 if replaced, or ENOMEM.

int **map_del** (*map_t* * *map*, const char * *str*)
Deletes str from the map, returns 0 on suceess.

void **map_clear** (*map_t* * *map*)
Clears the given map.

int **map_walk_prefixed** (*map_t* * *map*, const char * *prefix*, int(**callback*)(const char *, void *, void *), void * *baton*)
Calls callback for all strings in map with the given prefix.

Returns value immediately if a callback returns nonzero.

Parameters

- `map`:
- `prefix`: required string prefix (empty => all strings)
- `callback`: callback parameters are (key, value, baton)
- `baton`: passed uservalue

struct map_t
#include <map.h> Main data structure.

Public Members

void* **root**

struct knot_mm* **pool**

set

A set abstraction implemented on top of map.

Example usage:

Note The API is based on map.h, see it for more examples.

```
set_t set = set_make(NULL);

// Insert keys
if (set_add(&set, "princess") != 0 ||
    set_add(&set, "prince")   != 0 ||
    set_add(&set, "leia")     != 0) {
    fail();
}

// Test membership
if (set_contains(&set, "leia")) {
    success();
}

// Prefix search
int i = 0;
int count(const char *s, void *n) { (*(int *)n)++; return 0; }
if (set_walk_prefixed(set, "princ", count, &i) == 0) {
```

(continues on next page)

(continued from previous page)

```
    printf("%d matches\n", i);
}

// Delete
if (set_del(&set, "badkey") != 0) {
    fail(); // No such key
}

// Clear the set
set_clear(&set);
```

Defines

set_make
Creates an new, empty critbit set

set_contains(set, str)
Returns non-zero if set contains str

set_add(set, str)
Inserts str into set. Returns 0 if new, 1 if already present, or ENOMEM.

set_del(set, str)
Deletes str from the set, returns 0 on suceess

set_clear(set)
Clears the given set

set_walk(set, callback, baton)
Calls callback for all strings in map

set_walk_prefixed(set, prefix, callback, baton)
Calls callback for all strings in set with the given prefix

Typedefs

typedef *map_t* **set_t**

typedef int () set_walk_cb(const char *, void *)

pack

A length-prefixed list of objects, also an array list.

Each object is prefixed by item length, unlike array this structure permits variable-length data. It is also equivallent to forward-only list backed by an array.

Example usage:

Note Maximum object size is 2^16 bytes, see *pack_objlen_t* If some mistake happens somewhere, the access may end up in an infinite loop. (equality comparison on pointers)

```
pack_t pack;
pack_init(pack);

// Reserve 2 objects, 6 bytes total
```

(continues on next page)

(continued from previous page)

```
pack_reserve(pack, 2, 4 + 2);

// Push 2 objects
pack_obj_push(pack, U8("jedi"), 4)
pack_obj_push(pack, U8("\xbe\xef"), 2);

// Iterate length-value pairs
uint8_t *it = pack_head(pack);
while (it != pack_tail(pack)) {
    uint8_t *val = pack_obj_val(it);
    it = pack_obj_next(it);
}

// Remove object
pack_obj_del(pack, U8("jedi"), 4);

pack_clear(pack);
```

Defines

pack_init (pack)
Zero-initialize the pack.

pack_clear (pack)
Make the pack empty and free pointed-to memory (plain malloc/free).

pack_clear_mm (pack, free, baton)
Make the pack empty and free pointed-to memory.

Mempool usage: pass mm_free and a knot_mm_t* .

pack_reserve (pack, objs_count, objs_len)
Reserve space for *additional* objects in the pack (plain malloc/free).

Return 0 if success, <0 on failure

pack_reserve_mm (pack, objs_count, objs_len, reserve, baton)
Reserve space for *additional* objects in the pack.

Mempool usage: pass kr_memreserve and a knot_mm_t* .

Return 0 if success, <0 on failure

pack_head (pack)
Return pointer to first packed object.

Recommended way to iterate: for (uint8_t *it = *pack_head(pack)*; it != *pack_tail(pack)*; it = pack_obj_next(it))

pack_tail (pack)
Return pack end pointer.

Typedefs

typedef uint16_t **pack_objlen_t**
Packed object length type.

Functions

typedef **array_t** (uint8_t)
Pack is defined as an array of bytes.

static *pack_objlen_t* **pack_obj_len** (uint8_t * *it*)
Return packed object length.

static uint8_t* **pack_obj_val** (uint8_t * *it*)
Return packed object value.

static uint8_t* **pack_obj_next** (uint8_t * *it*)
Return pointer to next packed object.

static uint8_t* **pack_last** (pack_t *pack*)
Return pointer to the last packed object.

static int **pack_obj_push** (pack_t * *pack*, const uint8_t * *obj*, *pack_objlen_t* *len*)
Push object to the end of the pack.

Return 0 on success, negative number on failure

static uint8_t* **pack_obj_find** (pack_t * *pack*, const uint8_t * *obj*, *pack_objlen_t* *len*)
Returns a pointer to packed object.

Return pointer to packed object or NULL

static int **pack_obj_del** (pack_t * *pack*, const uint8_t * *obj*, *pack_objlen_t* *len*)
Delete object from the pack.

Return 0 on success, negative number on failure

static int **pack_clone** (pack_t ** *dst*, const pack_t * *src*, knot_mm_t * *pool*)
Clone a pack, replacing destination pack; (*dst == NULL) is valid input.

Return kr_error(ENOMEM) on allocation failure.

lru

A lossy cache.

Example usage:

```
// Define new LRU type
typedef lru_t(int) lru_int_t;

// Create LRU
lru_int_t *lru;
lru_create(&lru, 5, NULL, NULL);

// Insert some values
int *pi = lru_get_new(lru, "luke", strlen("luke"), NULL);
if (pi)
    *pi = 42;
pi = lru_get_new(lru, "leia", strlen("leia"), NULL);
if (pi)
    *pi = 24;
```

(continues on next page)

(continued from previous page)

```
// Retrieve values
int *ret = lru_get_try(lru, "luke", strlen("luke"), NULL);
if (!ret) printf("luke dropped out!\n");
    else  printf("luke's number is %d\n", *ret);

char *enemies[] = {"goro", "raiden", "subzero", "scorpion"};
for (int i = 0; i < 4; ++i) {
    int *val = lru_get_new(lru, enemies[i], strlen(enemies[i]), NULL);
    if (val)
        *val = i;
}

// We're done
lru_free(lru);
```

Note The implementation tries to keep frequent keys and avoid others, even if "used recently", so it may refuse to store it on *lru_get_new()*. It uses hashing to split the problem pseudo-randomly into smaller groups, and within each it tries to approximate relative usage counts of several most frequent keys/hashes. This tracking is done for *more* keys than those that are actually stored.

Defines

lru_t (type)
The type for LRU, parametrized by value type.

lru_create (ptable, max_slots, mm_ctx_array, mm_ctx)
Allocate and initialize an LRU with default associativity.

The real limit on the number of slots can be a bit larger but less than double.

Note The pointers to memory contexts need to remain valid during the whole life of the structure (or be NULL).

Parameters

- `ptable`: pointer to a pointer to the LRU
- `max_slots`: number of slots
- `mm_ctx_array`: memory context to use for the huge array, NULL for default If you pass your own, it needs to produce CACHE_ALIGNED allocations (ubsan).
- `mm_ctx`: memory context to use for individual key-value pairs, NULL for default

lru_free (table)
Free an LRU created by lru_create (it can be NULL).

lru_reset (table)
Reset an LRU to the empty state (but preserve any settings).

lru_get_try (table, key_, len_)
Find key in the LRU and return pointer to the corresponding value.

Return pointer to data or NULL if not found

Parameters

- `table`: pointer to LRU

- `key_`: lookup key
- `len_`: key length

lru_get_new(table, key_, len_, is_new)
Return pointer to value, inserting if needed (zeroed).

Return pointer to data or NULL (can be even if memory could be allocated!)

Parameters

- `table`: pointer to LRU
- `key_`: lookup key
- `len_`: key lengthkeys
- `is_new`: pointer to bool to store result of operation (true if entry is newly added, false otherwise; can be NULL).

lru_apply(table, function, baton)
Apply a function to every item in LRU.

Parameters

- `table`: pointer to LRU
- `function`: enum lru_apply_do (*function)(const char *key, uint len, val_type *val, void *baton) See enum lru_apply_do for the return type meanings.
- `baton`: extra pointer passed to each function invocation

lru_capacity(table)
Return the real capacity - maximum number of keys holdable within.

Parameters

- `table`: pointer to LRU

Enums

lru_apply_do
Possible actions to do with an element.

Values:

LRU_APPLY_DO_NOTHING

LRU_APPLY_DO_EVICT

trie

Typedefs

typedef void* **trie_val_t**
Native API of QP-tries:

- keys are char strings, not necessarily zero-terminated, the structure copies the contents of the passed keys

- values are void* pointers, typically you get an ephemeral pointer to it
- key lengths are limited by 2^32-1 ATM

XXX EDITORS: trie.{h,c} are synced from https://gitlab.labs.nic.cz/knot/knot-dns/tree/68352fc969/src/contrib/qp-trie only with tiny adjustments, mostly #includes and KR_EXPORT.

Element value.

typedef struct trie **trie_t**
Opaque structure holding a QP-trie.

typedef struct trie_it **trie_it_t**
Opaque type for holding a QP-trie iterator.

Functions

KR_EXPORT *trie_t** **trie_create** (knot_mm_t * *mm*)
Create a trie instance. Pass NULL to use malloc+free.

KR_EXPORT void **trie_free** (*trie_t* * *tbl*)
Free a trie instance.

KR_EXPORT void **trie_clear** (*trie_t* * *tbl*)
Clear a trie instance (make it empty).

KR_EXPORT size_t **trie_weight** (const *trie_t* * *tbl*)
Return the number of keys in the trie.

KR_EXPORT *trie_val_t** **trie_get_try** (*trie_t* * *tbl*, const char * *key*, *uint32_t* *len*)
Search the trie, returning NULL on failure.

KR_EXPORT *trie_val_t** **trie_get_first** (*trie_t* * *tbl*, char ** *key*, *uint32_t* * *len*)
Return pointer to the minimum. Optionally with key and its length.

KR_EXPORT *trie_val_t** **trie_get_ins** (*trie_t* * *tbl*, const char * *key*, *uint32_t* *len*)
Search the trie, inserting NULL trie_val_t on failure.

KR_EXPORT int **trie_get_leq** (*trie_t* * *tbl*, const char * *key*, *uint32_t* *len*, *trie_val_t* ** *val*)
Search for less-or-equal element.

Return KNOT_EOK for exact match, 1 for previous, KNOT_ENOENT for not-found, or KNOT_E*.

Parameters

- `tbl`: Trie.
- `key`: Searched key.
- `len`: Key length.
- `val`: Must be valid; it will be set to NULL if not found or errored.

KR_EXPORT int **trie_apply** (*trie_t* * *tbl*, int(**f*)(*trie_val_t* *, void *), void * *d*)
Apply a function to every trie_val_t, in order.

Return First nonzero from f() or zero (i.e. KNOT_EOK).

Parameters

- `d`: Parameter passed as the second argument to f().

KR_EXPORT int **trie_del** (*trie_t* * *tbl*, const char * *key*, *uint32_t* *len*, *trie_val_t* * *val*)
Remove an item, returning KNOT_EOK if succeeded or KNOT_ENOENT if not found.

If val!=NULL and deletion succeeded, the deleted value is set.

KR_EXPORT int **trie_del_first** (*trie_t* * *tbl*, char * *key*, *uint32_t* * *len*, *trie_val_t* * *val*)
Remove the first item, returning KNOT_EOK on success.

You may optionally get the key and/or value. The key is copied, so you need to pass sufficient len, otherwise kr_error(ENOSPC) is returned.

KR_EXPORT *trie_it_t** **trie_it_begin** (*trie_t* * *tbl*)
Create a new iterator pointing to the first element (if any).

KR_EXPORT void **trie_it_next** (*trie_it_t* * *it*)
Advance the iterator to the next element.

Iteration is in ascending lexicographical order. In particular, the empty string would be considered as the very first.

Note You may not use this function if the trie's key-set has been modified during the lifetime of the iterator (modifying values only is OK).

KR_EXPORT bool **trie_it_finished** (*trie_it_t* * *it*)
Test if the iterator has gone past the last element.

KR_EXPORT void **trie_it_free** (*trie_it_t* * *it*)
Free any resources of the iterator. It's OK to call it on NULL.

KR_EXPORT const char* **trie_it_key** (*trie_it_t* * *it*, size_t * *len*)
Return pointer to the key of the current element.

Note The optional len is uint32_t internally but size_t is better for our usage, as it is without an additional type conversion.

KR_EXPORT *trie_val_t** **trie_it_val** (*trie_it_t* * *it*)
Return pointer to the value of the current element (writable).

CHAPTER 17

Modules API reference

17.1 Supported languages

Currently modules written in C and Lua(JIT) are supported.

17.2 The anatomy of an extension

A module is a shared object or script defining specific functions/fields; here's an overview.

C	Lua	Params	Comment
`X_api()`[1]			API version
`X_init()`	`X.init()`	`module`	Constructor
`X_deinit()`	`X.deinit()`	`module`	Destructor
`X_config()`	`X.config()`	`module, str`	Configuration
`X_layer`	`X.layer`		*Module layer*
`X_props`			List of properties

The `X` corresponds to the module name; if the module name is `hints`, the prefix for constructor would be `hints_init()`. More details are in docs for the *`kr_module`* and *`kr_layer_api`* structures.

Note: The modules get ordered – by default in the same as the order in which they were loaded. The loading command can specify where in the order the module should be positioned.

17.3 Writing a module in Lua

The probably most convenient way of writing modules is Lua since you can use already installed modules from system and have first-class access to the scripting engine. You can also tap to all the events, that the C API has access to, but keep in mind that transitioning from the C to Lua function is slower than the other way round, especially when JIT-compilation is taken into account.

Note: The Lua functions retrieve an additional first parameter compared to the C counterparts - a "state". Most useful C functions and structures have lua FFI wrappers, sometimes with extra sugar.

The modules follow the Lua way, where the module interface is returned in a named table.

```
--- @module Count incoming queries
local counter = {}

function counter.init(module)
        counter.total = 0
        counter.last = 0
        counter.failed = 0
end

function counter.deinit(module)
        print('counted', counter.total, 'queries')
end

-- @function Run the q/s counter with given interval.
function counter.config(conf)
        -- We can use the scripting facilities here
        if counter.ev then event.cancel(counter.ev)
        event.recurrent(conf.interval, function ()
                print(counter.total - counter.last, 'q/s')
                counter.last = counter.total
        end)
end

return counter
```

The created module can be then loaded just like any other module, except it isn't very useful since it doesn't provide any layer to capture events. The Lua module can however provide a processing layer, just *like its C counterpart*.

```
-- Notice it isn't a function, but a table of functions
counter.layer = {
        begin = function (state, data)
                        counter.total = counter.total + 1
```

(continues on next page)

[1] Mandatory symbol; defined by using *`KR_MODULE_EXPORT()`*.

(continued from previous page)

```
                        return state
                end,
        finish = function (state, req, answer)
                        if state == kres.FAIL then
                                counter.failed = counter.failed + 1
                        end
                        return state
                end
}
```

There is currently an additional "feature" in comparison to C layer functions: some functions do not get called at all if `state == kres.FAIL`; see docs for details: *kr_layer_api*.

Since the modules are like any other Lua modules, you can interact with them through the CLI and and any interface.

Tip: Module discovery: `kres_modules.` is prepended to the module name and lua search path is used on that.

17.4 Writing a module in C

As almost all the functions are optional, the minimal module looks like this:

```
#include "lib/module.h"
/* Convenience macro to declare module ABI. */
KR_MODULE_EXPORT(mymodule)
```

Let's define an observer thread for the module as well. It's going to be stub for the sake of brevity, but you can for example create a condition, and notify the thread from query processing by declaring module layer (see the *Writing layers*).

```
static void* observe(void *arg)
{
        /* ... do some observing ... */
}

int mymodule_init(struct kr_module *module)
{
        /* Create a thread and start it in the background. */
        pthread_t thr_id;
        int ret = pthread_create(&thr_id, NULL, &observe, NULL);
        if (ret != 0) {
                return kr_error(errno);
        }

        /* Keep it in the thread */
        module->data = thr_id;
        return kr_ok();
}

int mymodule_deinit(struct kr_module *module)
{
        /* ... signalize cancellation ... */
        void *res = NULL;
        pthread_t thr_id = (pthread_t) module->data;
```

(continues on next page)

(continued from previous page)

```
        int ret = pthread_join(thr_id, res);
        if (ret != 0) {
                return kr_error(errno);
        }

        return kr_ok();
}
```

This example shows how a module can run in the background, this enables you to, for example, observe and publish data about query resolution.

17.5 Configuring modules

There is a callback `X_config()` that you can implement, see hints module.

17.6 Exposing C module properties

A module can offer NULL-terminated list of *properties*, each property is essentially a callable with free-form JSON input/output. JSON was chosen as an interchangeable format that doesn't require any schema beforehand, so you can do two things - query the module properties from external applications or between modules (e.g. *statistics* module can query *cache* module for memory usage). JSON was chosen not because it's the most efficient protocol, but because it's easy to read and write and interface to outside world.

Note: The `void *env` is a generic module interface. Since we're implementing daemon modules, the pointer can be cast to `struct engine*`. This is guaranteed by the implemented API version (see *Writing a module in C*).

Here's an example how a module can expose its property:

```
char* get_size(void *env, struct kr_module *m,
               const char *args)
{
        /* Get cache from engine. */
        struct engine *engine = env;
        struct kr_cache *cache = &engine->resolver.cache;
        /* Read item count */
        int count = (cache->api)->count(cache->db);
        char *result = NULL;
        asprintf(&result, "{ \"result\": %d }", count);

        return result;
}

struct kr_prop *cache_props(void)
{
        static struct kr_prop prop_list[] = {
                /* Callback,   Name,   Description */
                {&get_size, "get_size", "Return number of records."},
                {NULL, NULL, NULL}
        };
        return prop_list;
```

(continues on next page)

(continued from previous page)

```
}

KR_MODULE_EXPORT(cache)
```

Once you load the module, you can call the module property from the interactive console. *Note:* the JSON output will be transparently converted to Lua tables.

```
$ kresd
...
[system] started in interactive mode, type 'help()'
> modules.load('cached')
> cached.get_size()
[size] => 53
```

17.6.1 Special properties

If the module declares properties `get` or `set`, they can be used in the Lua interpreter as regular tables.

CHAPTER 18

Worker API reference

Functions

int **worker_init** (struct engine * *engine*, int *worker_id*, int *worker_count*)
Create and initialize the worker.

Return error code (ENOMEM)

void **worker_deinit** (void)
Destroy the worker (free memory).

int **worker_submit** (struct session * *session*, const struct sockaddr * *peer*, knot_pkt_t * *query*)
Process an incoming packet (query from a client or answer from upstream).

Return 0 or an error code

Parameters

- `session`: session the packet came from
- `peer`: address the packet came from
- `query`: the packet, or NULL on an error from the transport layer

int **worker_end_tcp** (struct session * *session*)
End current DNS/TCP session, this disassociates pending tasks from this session which may be freely closed afterwards.

KR_EXPORT knot_pkt_t* **worker_resolve_mk_pkt** (const char * *qname_str*, uint16_t *qtype*, uint16_t *qclass*, const struct *kr_qflags* * *options*)
Create a packet suitable for worker_resolve_start().

All in malloc() memory.

KR_EXPORT struct qr_task* **worker_resolve_start** (knot_pkt_t * *query*, struct *kr_qflags* *options*)
Start query resolution with given query.

Return task or NULL

KR_EXPORT int **worker_resolve_exec** (struct qr_task * *task*, knot_pkt_t * *query*)

struct *kr_request** **worker_task_request** (struct qr_task * *task*) **Return**
struct *kr_request* associated with opaque task

int **worker_task_step** (struct qr_task * *task*, const struct sockaddr * *packet_source*, knot_pkt_t * *packet*)

int **worker_task_numrefs** (const struct qr_task * *task*)

int **worker_task_finalize** (struct qr_task * *task*, int *state*)
Finalize given task.

void **worker_task_complete** (struct qr_task * *task*)

void **worker_task_ref** (struct qr_task * *task*)

void **worker_task_unref** (struct qr_task * *task*)

void **worker_task_timeout_inc** (struct qr_task * *task*)

int **worker_add_tcp_connected** (struct worker_ctx * *worker*, const struct sockaddr * *addr*, struct session * *session*)

int **worker_del_tcp_connected** (struct worker_ctx * *worker*, const struct sockaddr * *addr*)

int **worker_del_tcp_waiting** (struct worker_ctx * *worker*, const struct sockaddr * *addr*)

knot_pkt_t* **worker_task_get_pktbuf** (const struct qr_task * *task*)

struct request_ctx* **worker_task_get_request** (struct qr_task * *task*)

struct session* **worker_request_get_source_session** (struct request_ctx *)

void **worker_request_set_source_session** (struct request_ctx *, struct session * *session*)

uint16_t **worker_task_pkt_get_msgid** (struct qr_task * *task*)

void **worker_task_pkt_set_msgid** (struct qr_task * *task*, uint16_t *msgid*)

uint64_t **worker_task_creation_time** (struct qr_task * *task*)

void **worker_task_subreq_finalize** (struct qr_task * *task*)

bool **worker_task_finished** (struct qr_task * *task*)

int **qr_task_on_send** (struct qr_task * *task*, uv_handle_t * *handle*, int *status*)
To be called after sending a DNS message.

It mainly deals with cleanups.

Variables

KR_EXPORT struct worker_ctx* **the_worker**
Pointer to the singleton worker.

NULL if not initialized.

struct worker_stats
#include <worker.h> Various worker statistics.

Sync with wrk_stats()

Public Members

size_t **queries**
: Total number of requests (from clients and internal ones).

size_t **concurrent**
: The number of requests currently in processing.

size_t **rconcurrent**

size_t **dropped**
: The number of requests dropped due to being badly formed.

  See #471.

size_t **timeout**
: Number of outbound queries that timed out.

size_t **udp**
: Number of outbound queries over UDP.

size_t **tcp**
: Number of outbound queries over TCP (excluding TLS).

size_t **tls**
: Number of outbound queries over TLS.

size_t **ipv4**
: Number of outbound queries over IPv4.

size_t **ipv6**
: Number of outbound queries over IPv6.

CHAPTER 19

Indices and tables

- genindex
- modindex
- search

Index

L

M

V

W