Knot Resolver

Release 6.0.0a1

CZ.NIC Labs

Jun 05, 2023

GETTING STARTED

Welcome to Knot Resolver's documentation! Knot Resolver is an opensource implementation of a caching validating DNS resolver. Modular architecture keeps the core tiny and efficient, and it also provides a state-machine like API for extensions.

If you are a new user, please start with chapter for getting started.

CHAPTER

ONE

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.

Note: Please note that the packages available in distribution repositories of Debian and Ubuntu are outdated.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.

Debian/Ubuntu

```
$ 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 epel-release
$ sudo yum install -y knot-resolver
```

Fedora

```
$ sudo dnf install -y knot-resolver
```

Arch Linux

```
$ sudo pacman -S knot-resolver
```

openSUSE Leap/Tumbleweed

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

Note: If for some reason you need to **install Knot Resolver from source**, check out *building from sources* documentation for developers.

CHAPTER

TWO

STARTUP

The main way to run Knot Resolver is to use provided integration with `systemd`.

```
$ sudo systemctl start knot-resolver.service
```

See logs and status of running instance with `systemctl status knot-resolver.service` command. For more information about systemd integration see `man knot-resolver.systemd`.

Warning: `knot-resolver.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 knot-resolver.service`

Unfortunately, for some cases (typically Docker and minimalistic systems), `systemd` is not available, therefore it is not possible to use `knot-resolver.service`. If you have this problem, look at usage without systemd section.

Note: If for some reason you need to use Knot Resolver as it was before version 6, check out *usage without the manager* Otherwise, it is recommended to stick to this chapter.

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

THREE

CONFIGURATION

Easiest way to configure Knot Resolver is to put YAML configuration in `/etc/knot-resolver/config.yml` file.

You can start exploring the configuration by continuing in this chapter or look at the complete *configuration* documentation.

Complete examples of configuration files can be found here. Examples are also installed as documentation files, typically in `/usr/share/doc/knot-resolver/examples/` directory (location may be different based on your Linux distribution).

Tip: You can use *kresctl* utility to **validate** your configuration before pushing it into the running resolver. It should help prevent many typos in the configuration.

```
$ kresctl validate /etc/knot-resolver/config.yml
```

If you update the configuration file while Knot Resolver is running, you can force the resolver to **reload** it by invoking a `systemd` reload command.

```
$ systemctl reload knot-resolver.service
```

Note: **Reloading configuration** can fail even when your configuration is valid, because some options cannot be changed while running. You can always find an explanation of the error in the log accesed by the `journalctl -eu knot-resolver` command.

3.1 Listening on network interfaces

The first thing you will probably want to configure are the network interfaces to listen to. The following example instructs the resolver to receive standard unencrypted DNS queries on `192.0.2.1` and `2001:db8::1` IP addresses. Encrypted DNS queries using `DNS-over-TLS` protocol are accepted on all IP addresses of `eth0` network interface, TCP port `853`.

```
network:
    listen:
    - interface: ['192.0.2.1', '2001:db8::1'] # port 53 is default
    - interface: 'eth0'
        port: 853
        kind: 'dot' # DNS-over-TLS
```

For more details look at the network configuration.

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 Example: 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`.

```
policy:
```

See chapter *Replacing part of the DNS tree* for more details.

3.3 Example: 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 example.

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 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.

```
view:

policy:
```

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:

```
network:
    listen:
    - interface: ['192.0.2.1', '2001:db8::1']
        kind: 'dot' # DNS-over-TLS, port 853 is default
```

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:

```
network:
    tls:
        cert-file: '/etc/knot-resolver/server-cert.pem'
        key-file: '/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:
```

3.4 Example: 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 *TLS forwarding* which provides methods for authentication. .. It can be configured using *policy.TLS_FORWARD* 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 slice function, it is possible to split the .. With the use of `policy.slice` function, it is possible to split the entire DNS namespace into distinct "slices". When used in conjunction with *TLS forwarding*, it's possible to forward different queries to different .. *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 encrypt **all** your traffic and not just DNS queries!

```
policy:
    # TODO
```

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*.

CHAPTER

FOUR

CONFIGURATION OVERVIEW

Configuration file is by default named `/etc/knot-resolver/config.yml`. Different configuration file can be loaded by using command line option `-c / --config`.

4.1 Syntax

The configuration file uses YAML format version 1.1. To quickly learn about the format, you can have a look at Learn YAML in Y minutes.

4.2 Schema

The configuration has to pass a validation step before being used. The validation mainly checks for conformance to our configuration-schema.

Tip: Whenever a configuration is loaded and the validation fails, we attempt to log a detailed error message explaining what the problem was. For example, it could look like the following:

If you happen to find a rejected configuration with unhelpful or confusing error message, please report it as a bug.

Tip: An easy way to see the complete configuration structure is to look at the JSON schema represention. The raw JSON schema is available at this link (valid only for the version of resolver this documentation was generated for). For the schema readability, some graphical visualizer can be used, for example this one.

CHAPTER

FIVE

CONFIGURATION SCHEMA

The configuration schema describes the structure of accepted configuration files (or objects via the API). While originally specified in Python source code, it can be visualized as a JSON schema.

5.1 Getting the JSON schema

1. The JSON schema can be obtained from a running Resolver by sending a HTTP GET request to the path `/schema` on the management socket (by default a Unix socket at `/var/run/knot-resolver/manager.sock`).
2. The `kresctl schema` command outputs the schema of the currently installed version as well. It does not require a running resolver.
3. JSON schema for the most recent Knot Resolver version can be downloaded here.

5.2 Validating you configuration

As mentioned above, the JSON schema is NOT used to validate the configuration in the Knot Resolver. It's the other way around, the validation process can generate JSON schema that can help you understand the configuration structure. Some validation steps are however dynamic (for example resolving of interface names) and can not be expressed using JSON schema and cannot be even completed without running full Resolver.

Note: When using the API to change configuration in runtime, your change can be rejected by the validation step even though Knot Resolver would start just fine with the given changed configuration. Some validation steps within the Resolver are dynamic and they are dependent on both your previous configuration and the new one. For example, if you try to change the management socket, the validation will fail even though the new provided address is perfectly valid. Chaning the management socket while running is not supported.

Most of the validation is however static and you can use the `kresctl validate` command to check your configuration file for most errors before actually running the Resolver.

5.3 Interactive visualization

The following visualization is interactive and offers good overview of the configuration structure.

5.4 Text-based configuration schema description

Following, you can find the JSON schema flattened textual representation. It's not meant to be read top-to-bottom, however it can be used as a quick lookup reference.

CHAPTER

SIX

LISTENING ON NETWORK INTERFACES

The first thing you will probably need to configure are the network interfaces to listen to.

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.

YAML

```
network:
  listen:
    - interface: ['192.0.2.1', '2001:db8::1'] # unencrypted DNS on port 53 is default
    - interface: 'eth0'
      port: 853
      kind: 'dot'
```

Lua

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

```
-- 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.

CHAPTER

SEVEN

ADVANCED CONFIGURATION (LUA)

Knot Resolver can be configured declaratively by using YAML files or YAML/JSON HTTP API. However, there is another option. The actual worker processes (the `kresd` executable) speaks a different configuration language, it internally uses the Lua runtime and the respective programming language.

Essentially, the declarative configuration is only used for validation and as an external interface. After validation, a Lua configuration is generated and passed into individual `kresd` instances. You can see the generated configuration files within the Resolver's working directory or you can manually run the conversion of declarative configuration with the `kresctl convert` command.

Warning: While there are no plans of ever removing the Lua configuration, we do not guarantee absence of backwards incompatible changes. Starting with Knot Resolver version 6 and later, we consider the Lua interface internal and a subject to change. While we don't have any breaking changes planned for the foreseeable future, they might come.

Therefore, use this only when you don't have any other option. And please let us know about it and we might try to accomodate your usecase in the declarative configuration.

7.1 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.

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.

7.2 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 interactively and its output
> log_level()
'notice'
> -- the previous line without > character is output from log_level() command
```

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

```
> modules.list()
{
    'iterate',
    'validate',
    'cache',
    'ta_update',
    'ta_signal_query',
    'policy',
    'priming',
    'detect_time_skew',
    'detect_time_jump',
    'ta_sentinel',
    'edns_keepalive',
    'refuse_nord',
    'watchdog',
}
```

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

7.3 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!

7.4 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.

7.4.1 Server (communication with clients)

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 (unencrypted UDP, *using XDP Linux API*)	`xdp`
DNS-over-TLS (DoT)	`tls`
DNS-over-HTTPS (DoH)	`doh2`
Web management	`webmgmt`
Control socket	`control`
Legacy DNS-over-HTTPS (DoH)	`doh_legacy`

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 (UDP, *using XDP*)	`net.listen('192.0.2.123', 53, { kind = 'xdp' })`
DNS-over-TLS (DoT)	`net.listen('192.0.2.123', 853, { kind = 'tls' })`
DNS-over-HTTPS (DoH)	`net.listen('192.0.2.123', 443, { kind = 'doh2' })`
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 = 'doh2' })
net.listen('::', 8453, { kind = 'webmgmt' }) -- see http module
net.listen('/tmp/kresd-socket', nil, { kind = 'webmgmt' }) -- http module␣
↪supports AF_UNIX
net.listen('eth0', 53, { kind = 'xdp' })
net.listen('192.0.2.123', 53, { kind = 'xdp', nic_queue = 0 })
```

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.

PROXYv2 protocol

Knot Resolver supports proxies that utilize the PROXYv2 protocol to identify clients.

A PROXY header contains the IP address of the original client who sent a query. This allows the resolver to treat queries as if they actually came from the client's IP address rather than the address of the proxy they came through. For example, *Views and ACLs* are able to work properly when PROXYv2 is in use.

Since allowing usage of the PROXYv2 protocol for all clients would be a security vulnerability, because clients would then be able to spoof their IP addresses via the PROXYv2 header, the resolver requires you to specify explicitly which clients are allowed to send PROXYv2 headers via the *net.proxy_allowed()* function.

PROXYv2 queries from clients who are not explicitly allowed to use this protocol will be discarded.

`net.`**`proxy_allowed`**`(`[*addresses*]`)`

Allow usage of the PROXYv2 protocol headers by clients on the specified `addresses`. It is possible to permit whole networks to send PROXYv2 headers by specifying the network mask using the CIDR notation (e.g. `172.22.0.0/16`). IPv4 as well as IPv6 addresses are supported.

If you wish to allow all clients to use PROXYv2 (e.g. because you have this kind of security handled on another layer of your network infrastructure), you can specify a netmask of `/0`. Please note that this setting is address-family-specific, so this needs to be applied to both IPv4 and IPv6 separately.

Subsequent calls to the function overwrite the effects of all previous calls. Providing a table of strings as the function parameter allows multiple distinct addresses to use the PROXYv2 protocol.

When called without arguments, `net.proxy_allowed` returns a table of all addresses currently allowed to use the PROXYv2 protocol and does not change the configuration.

Examples:

```
net.proxy_allowed('172.22.0.1')    -- allows '172.22.0.1' specifically
net.proxy_allowed('172.18.1.0/24') -- allows everyone at '172.18.1.*'
net.proxy_allowed({
        '172.22.0.1', '172.18.1.0/24'
})                                 -- allows both of the above at once
net.proxy_allowed({ 'fe80::/10' }  -- allows everyone at IPv6 link-local
net.proxy_allowed({
```

(continues on next page)

(continued from previous page)

```
        '::/0', '0.0.0.0/0'
})                                  -- allows everyone
net.proxy_allowed('::/0')           -- allows all IPv6 (but no IPv4)
net.proxy_allowed({})               -- prevents everyone from using PROXYv2
net.proxy_allowed()                 -- returns a list of all currently allowed␣
↪addresses
```

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
    }
}
[4] => {
```

(continues on next page)

(continued from previous page)

```
    [kind] => xdp
    [transport] => {
        [family] => inet4+inet6
        [interface] => eth2
        [nic_queue] => 0
        [port] => 53
        [protocol] => udp
    }
}
```

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
    }
    [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.

DoT and DoH (encrypted DNS)

Warning: It is important to understand **limits of encrypting only DNS traffic**. Relevant security analysis can be found in article *Simran Patil and Nikita Borisov. 2019. What can you learn from an IP?* See slides or the article itself.

DoT and DoH encrypt DNS traffic with Transport Layer Security (TLS) protocol and thus protects DNS traffic from certain types of attacks.

You can learn more about DoT and DoH and their implementation in Knot Resolver in this article.

DNS-over-TLS (DoT)

DNS-over-TLS server (**RFC 7858**) can be configured using `tls` kind in *`net.listen()`*. It is enabled on localhost by default.

For certificate configuration, refer to *HTTP status codes*.

DNS-over-HTTPS (DoH)

Note: Knot Resolver currently offers two DoH implementations. It is recommended to use this new implementation, which is more reliable, scalable and has fewer dependencies. Make sure to use `doh2` kind in *`net.listen()`* to select this implementation.

Tip: Independent information about political controversies around the DoH deployment by default can be found in blog posts DNS Privacy at IETF 104 and More DOH by Geoff Huston and Centralised DoH is bad for Privacy, in 2019 and beyond by Bert Hubert.

DNS-over-HTTPS server (**RFC 8484**) can be configured using `doh2` kind in *`net.listen()`*.

This implementation supports HTTP/2 (**RFC 7540**). Queries can be sent to the `/dns-query` endpoint, e.g.:

```
$ kdig @127.0.0.1 +https www.knot-resolver.cz AAAA
```

Only TLS version 1.3 (or higher) is supported with DNS-over-HTTPS. The additional considerations for TLS 1.2 required by HTTP/2 are not implemented (**RFC 7540#section-9.2**).

Warning: Take care when configuring your server to listen on well known HTTPS port. If an unrelated HTTPS service is running on the same port with REUSEPORT enabled, you will end up with both services malfunctioning.

HTTP status codes

As specified by RFC 8484, the resolver responds with status **200 OK** whenever it can produce a valid DNS reply for a given query, even in cases where the DNS `rcode` indicates an error (like `NXDOMAIN`, `SERVFAIL`, etc.).

For DoH queries malformed at the HTTP level, the resolver may respond with the following status codes:

- **400 Bad Request** for a generally malformed query, like one not containing a valid DNS packet
- **404 Not Found** when an incorrect HTTP endpoint is queried - the only supported ones are `/dns-query` and `/doh`
- **413 Payload Too Large** when the DNS query exceeds its maximum size
- **415 Unsupported Media Type** when the query's `Content-Type` header is not `application/dns-message`
- **431 Request Header Fields Too Large** when a header in the query is too large to process
- **501 Not Implemented** when the query uses a method other than `GET`, `POST`, or `HEAD`

Configuration options for DoT and DoH

Note: These settings affect both DNS-over-TLS and DNS-over-HTTPS (except the legacy implementation).

A self-signed certificate is generated by default. 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*])

When called with path arguments, the function loads the server TLS certificate and private key for DoT and DoH.

When called without arguments, the command returns the currently configured paths.

Example output:

```
> net.tls("/etc/knot-resolver/server-cert.pem", "/etc/knot-resolver/server-key.pem")
> net.tls()  -- print configured paths
[cert_file] => '/etc/knot-resolver/server-cert.pem'
[key_file] => '/etc/knot-resolver/server-key.pem'
```

Tip: The certificate files aren't automatically reloaded on change. If you update the certificate files, e.g. using ACME, you have to either restart the service(s) or call this function again using *Control sockets*.

`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 endianness 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 and GnuTLS < 3.7.5**. GnuTLS 3.7.5 adds an option to disable resumption via tickets for TLS <= 1.2, enabling them only for protocols that do guarantee PFS. Knot Resolver makes use of this new option when linked against GnuTLS >= 3.7.5.

`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.

`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.

Configuration options for DoH

`net.doh_headers`([*string or table of strings*])

Selects the headers to be exposed. These headers and their values are available in `request.qsource.headers`. Comparison is case-insensitive and pseudo-headers are supported as well.

The following snippet can be used in the lua module to access headers `:method` and `user-agent`:

```
net.doh_headers({':method', 'user-agent'})

...

for i = 1, tonumber(req.qsource.headers.len) do
  local name = ffi.string(req.qsource.headers.at[i - 1].name)
  local value = ffi.string(req.qsource.headers.at[i - 1].value)
  print(name, value)
end
```

Other 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.

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

Kind	Explanation
webmgmt	*built-in web management* APIs (includes DoH)
doh_legacy	*Legacy DNS-over-HTTPS (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',
        -- 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 unencrypted 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',
```

(continues on next page)

(continued from previous page)

```
        key  = '/etc/knot-resolver/mykey.key',
}, 'doh_legacy')
```

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.

Legacy DNS-over-HTTPS (DoH)

Warning: The legacy DoH implementation using `http` module (`kind='doh_legacy'`) is deprecated. It has known performance and stability issues that won't be fixed. Use new *DNS-over-HTTPS (DoH)* implementation instead.

This was an experimental implementation of **RFC 8484**. It can be configured using `doh_legacy` kind in *`net.listen()`*. Its configuration (such as certificates) takes place in `http.config()`.

Queries were served on `/doh` and `/dns-query` endpoints.

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 its debug-level logs.
`/doh`	Legacy DNS-over-HTTPS	**RFC 8484** endpoint, see *Legacy DNS-over-HTTPS (DoH)*.
`/dns-query`	Legacy DNS-over-HTTPS	**RFC 8484** endpoint, see *Legacy DNS-over-HTTPS (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
```

  Some other systems can install from LuaRocks directly:

```
$ luarocks --lua-version 5.1 install http
```

- (*optional*) mmdblua available in LuaRocks

```
$ luarocks --lua-version 5.1 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
```

7.4.2 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.

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, though the impact of IPv6 outage is lowered since release 5.3.0.

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.

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 answers 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*.

7.4.3 DNS protocol tweaks

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_downstream_bufsize][, udp_upstream_bufsize]*)

Get/set maximum EDNS payload size advertised in DNS packets. Different values can be configured for communication downstream (towards clients) and upstream (towards other DNS servers). Set and also get operations use values in this order.

Default is 1232 bytes which was chosen to minimize risk of issues caused by IP fragmentation. Further details can be found at DNS Flag Day 2020 web site.

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 by DNSSEC standard **RFC 4035**.

Example output:

```
-- set downstream and upstream bufsize to value 4096
> net.bufsize(4096)
-- get configured downstream and upstream bufsizes, respectively
> net.bufsize()
4096    -- result # 1
4096    -- result # 2

-- set downstream bufsize to 4096 and upstream bufsize to 1232
> net.bufsize(4096, 1232)
-- get configured downstream and upstream bufsizes, respectively
> net.bufsize()
4096    -- result # 1
1232    -- result # 2
```

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

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

7.5 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.

7.5.1 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.

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.

Note: Choosing a cache size that can fit into RAM is important even if the cache is stored on disk (default). Otherwise, the extra I/O caused by disk access for missing pages can cause performance issues.

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
```

It is also possible to set the cache size based on the file system size. This is useful if you use a dedicated partition for cache (e.g. non-persistent tmpfs). It is recommended to leave some free space for special files, such as locks.:

```
cache.size = cache.fssize() - 10*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.

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 as tmpfs. While it is technically possible to move the cache to an existing tmpfs filesystem, it is *not recommended*, since the path to cache is configured in multiple places.

Mounting the cache directory as tmpfs is the 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/kresd.conf
cache.size = cache.fssize() - 10*MB
```

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.**fssize**()

Returns
Partition size of cache storage.

cache.**stats**()

Return table with low-level statistics for internal cache operation and storage. 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()
[clear] => 0
[close] => 0
[commit] => 117
[count] => 2
[count_entries] => 6187
[match] => 21
[match_miss] => 2
[open] => 0
[read] => 4313
[read_leq] => 9
[read_leq_miss] => 4
```

(continues on next page)

(continued from previous page)

```
[read_miss] => 1143
[remove] => 17
[remove_miss] => 0
[usage_percent] => 15.625
[write] => 189
```

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 contains 6187 internal entries which occupy 15.625 % cache size.

cache.**max_ttl**([*ttl*])

Parameters
ttl (*number*) – maximum TTL in seconds (default: 1 day)

Returns
current maximum TTL

Get or set upper TTL bound applied to all received records.

Note: The *ttl* value must be in range *(min_ttl, 2147483647)*.

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

cache.**min_ttl**([*ttl*])

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

Returns
current minimum TTL

Get or set lower TTL bound applied to all received records. Forcing TTL higher than specified violates DNS standards, so use higher values with care. TTL still won't be extended beyond expiration of the corresponding DNSSEC signature.

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

```
-- 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

- **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.

[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.

- **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] asynchronous cache.clear('com', false) finished

-- Clear only 'www.example.com.'
> cache.clear('www.example.com.', true)
[round] => 1
[count] => 1
[not_apex] => to clear proofs of non-existence call cache.clear('example.com.')
[subtree] => example.com.
```

7.5.2 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 incoming 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)`.

Zero-downtime restarts

Resolver restart normally takes just milliseconds 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.

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' })
```

(continues on next page)

(continued from previous page)

```
elseif string.match(systemd_instance, '^doh') then
     net.listen('127.0.0.1', 443, { kind = 'doh2' })
else
     panic("Use kresd@dns*, kresd@tls* or kresd@doh* instance names")
end
```

7.5.3 Prefetching records

The `predict` module helps to keep the cache hot by prefetching records. It can utilize two independent mechanisms to select the records which should be refreshed: expiring records and prediction.

Expiring records

This mechanism is always active when the predict module is loaded and it is not configurable.

Any time the resolver answers with records that are about to expire, they get refreshed. (see *is_expiring()*) That improves latency for records which get frequently queried, relatively to their TTL.

Prediction

The predict module can also learn usage patterns and repetitive queries, though this mechanism is a prototype and **not recommended** for use in production or with high traffic.

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.

You can disable prediction by configuring `period = 0`. Otherwise it will load the required *stats* module if not present, and it will use its *stats.frequent()* table and clear it periodically.

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.

Example configuration

```
modules = {
        predict = {
                -- this mode is NOT RECOMMENDED for use in production
                window = 15, -- 15 minutes sampling window
                period = 6*(60/15) -- track last 6 hours
        }
}
```

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

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.

7.5.4 Cache prefilling

This module provides ability to periodically prefill the 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 servers.

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

The root zone to be imported must be signed using DNSSEC and the resolver must have a 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.

Dependencies

Prefilling depends on the lua-http library.

7.5.5 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*.

Running

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

7.5.6 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.

7.5.7 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; you may disable it by adding `modules.unload('priming')` to your configuration.

7.5.8 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.unload('edns_keepalive')
```

7.5.9 XDP for higher UDP performance

Warning: As of version 5.2.0, XDP support in Knot Resolver is considered experimental. The impact on overall throughput and performance may not always be beneficial.

Using XDP allows significant speedup of UDP packet processing in recent Linux kernels, especially with some network drivers that implement good support. The basic idea is that for selected packets the Linux networking stack is bypassed, and some drivers can even directly use the user-space buffers for reading and writing.

Prerequisites

Warning: Bypassing the network stack has significant implications, such as bypassing the firewall and monitoring solutions. Make sure you're familiar with the trade-offs before using this feature. Read more in *Limitations*.

- Linux kernel 4.18+ (5.x+ is recommended for optimal performance) compiled with the *CONFIG_XDP_SOCKETS=y* option. XDP isn't supported in other operating systems.
- libknot compiled with XDP support
- **A multiqueue network card with native XDP support is highly recommended**, otherwise the performance gain will be much lower and you may encounter issues due to XDP emulation. Successfully tested cards:
  - Intel series 700 (driver *i40e*), maximum number of queues per interface is 64.
  - Intel series 500 (driver *ixgbe*), maximum number of queues per interface is 64. The number of CPUs available has to be at most 64!

Set up

The server instances need additional Linux **capabilities** during startup. (Or you could start them as *root*.) Execute command

```
systemctl edit kresd@.service
```

And insert these lines:

```
[Service]
CapabilityBoundingSet=CAP_NET_RAW CAP_NET_ADMIN CAP_SYS_ADMIN CAP_IPC_LOCK CAP_SYS_
↪RESOURCE
AmbientCapabilities=CAP_NET_RAW CAP_NET_ADMIN CAP_SYS_ADMIN CAP_IPC_LOCK CAP_SYS_RESOURCE
```

The `CAP_SYS_RESOURCE` is only needed on Linux < 5.11.

You want the same number of kresd instances and network **queues** on your card; you can use `ethtool -L` before the services start. With XDP this is more important than with vanilla UDP, as we only support one instance per queue and unclaimed queues will fall back to vanilla UDP. Ideally you can set these numbers as high as the number of CPUs that you want kresd to use.

Modification of `/etc/knot-resolver/kresd.conf` may often be quite simple, for example:

```
net.listen('eth2', 53, { kind = 'xdp' })
net.listen('203.0.113.53', 53, { kind = 'dns' })
```

Note that you want to also keep the vanilla DNS line to service TCP and possibly any fallback UDP (e.g. from unclaimed queues). XDP listening is in principle done on queues of whole network interfaces and the target addresses of incoming packets aren't checked in any way, but you are still allowed to specify interface by an address (if it's unambiguous at that moment):

```
net.listen('203.0.113.53', 53, { kind = 'xdp' })
net.listen('203.0.113.53', 53, { kind = 'dns' })
```

The default selection of queues is tailored for the usual naming convention: `kresd@1.service`, `kresd@2.service`, ... but you can still specify them explicitly, e.g. the default is effectively the same as:

```
net.listen('eth2', 53, { kind = 'xdp', nic_queue = env.SYSTEMD_INSTANCE - 1 })
```

Optimizations

Some helpful commands:

```
ethtool -N <interface> rx-flow-hash udp4 sdfn
ethtool -N <interface> rx-flow-hash udp6 sdfn
ethtool -L <interface> combined <queue-number>
ethtool -G <interface> rx <ring-size> tx <ring-size>
renice -n 19 -p $(pgrep '^ksoftirqd/[0-9]*$')
```

Limitations

- VLAN segmentation is not supported.
- MTU higher than 1792 bytes is not supported.
- Multiple BPF filters per one network device are not supported.
- Symmetrical routing is required (query source MAC/IP addresses and reply destination MAC/IP addresses are the same).
- Systems with big-endian byte ordering require special recompilation of libknot.
- IPv4 header and UDP checksums are not verified on received DNS messages.
- DNS over XDP traffic is not visible to common system tools (e.g. firewall, tcpdump etc.).
- BPF filter is not automatically unloaded from the network device. Manual filter unload:

  ```
  ip link set dev <interface> xdp off
  ```

- Knot Resolver only supports using XDP towards clients currently (not towards upstreams).
- When starting up an XDP socket you may get a harmless warning:

  ```
  libbpf: Kernel error message: XDP program already attached
  ```

7.6 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.

7.6.1 Query policies

This module can block, rewrite, or alter inbound queries based on user-defined policies. It does not affect queries generated by the resolver itself, e.g. when following CNAME chains etc.

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 *`policy.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 examples below.

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 overridden by action *`policy.PASS`*. For debugging purposes you can also add `modules.unload('policy')` to your config to unload the module.

Filters

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

`policy.`**`all`**`(`*action*`)`

Always applies the action.

`policy.`**`pattern`**`(`*action*`, `*pattern*`)`

Applies the action if query name matches a Lua regular expression.

`policy.`**`suffix`**`(`*action*`, `*suffix_table*`)`

Applies the action if query name suffix matches one of suffixes in the table (useful for "is domain in zone" rules).

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

Note: For speed this filter requires domain names in DNS wire format, not textual representation, so each label in the name must be prefixed with its length. Always use convenience function *policy.todnames()* for automatic conversion from strings! For example:

Note: Non-ASCII is not supported.

Knot Resolver does not provide any convenience support for IDN. Therefore everywhere (all configuration, logs, RPZ files) you need to deal with the xn-- forms of domain name labels, instead of directly using unicode characters.

`policy.`**`domains`**(*action*, *domain_table*)

Like *policy.suffix()* match, but the queried name must match exactly, not just its suffix.

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

Parameters

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

Like *policy.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").

It is also possible to define custom filter function with any name.

`policy.`**`custom_filter`**(*state*, *query*)

Parameters

- **state** – Request processing state *kr_layer_state*, typically not used by filter function.
- **query** – Incoming DNS query as *kr_query* structure.

Returns

An *action* function or `nil` if filter did not match.

Typically filter function is generated by another function, which allows easy parametrization - this technique is called closure. An practical example of such filter generator is:

```
function match_query_type(action, target_qtype)
    return function (state, query)
        if query.stype == target_qtype then
            -- filter matched the query, return action function
            return action
        else
            -- filter did not match, continue with next filter
            return nil
        end
    end
end
```

This custom filter can be used as any other built-in filter. For example this applies our custom filter and executes action *policy.DENY* on all queries of type *HINFO*:

```
-- custom filter which matches HINFO queries, action is policy.DENY
policy.add(match_query_type(policy.DENY, kres.type.HINFO))
```

Actions

An *action* is a function which modifies DNS request, and is either of type *chain* or *non-chain*:

- *Non-chain actions* modify state of the request and stop rule processing. An example of such action is *Forwarding*.
- *Chain actions* modify state of the request and allow other rules to evaluate and act on the same request. One such example is *policy.MIRROR()*.

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:

`policy.`**`PASS`**

Let the query pass through; it's useful to make exceptions before wider rules. For example:

More specific whitelist rule must precede generic blacklist rule:

```
-- Whitelist 'good.example.com'
policy.add(policy.pattern(policy.PASS, todname('good.example.com.')))
-- Block all names below example.com
policy.add(policy.suffix(policy.DENY, {todname('example.com.')}))
```

`policy.`**`DENY`**

Deny existence of names matching filter, i.e. reply NXDOMAIN authoritatively.

`policy.`**`DENY_MSG`**(*message*[, *extended_error=kres.extended_error.BLOCKED*])

Deny existence of a given domain and add explanatory message. NXDOMAIN reply contains an additional explanatory message as TXT record in the additional section.

You may override the extended DNS error to provide the user with more information. By default, `BLOCKED` is returned to indicate the domain is blocked due to the internal policy of the operator. Other suitable error codes are `CENSORED` (for externally imposed policy reasons) or `FILTERED` (for blocking requested by the client). For more information, please refer to **RFC 8914**.

`policy.`**`DROP`**

Terminate query resolution and return SERVFAIL to the requestor.

`policy.`**`REFUSE`**

Terminate query resolution and return REFUSED to the requestor.

`policy.`**`NO_ANSWER`**

Terminate query resolution and do not return any answer to the requestor.

> **Warning:** During normal operation, an answer should always be returned. Deliberate query drops are indistinguishable from packet loss and may cause problems as described in **RFC 8906**. Only use *NO_ANSWER* on very specific occasions, e.g. as a defense mechanism during an attack, and prefer other actions (e.g. *DROP* or *REFUSE*) for normal operation.

`policy.TC`

Force requestor to use TCP. It sets truncated bit (*TC*) in response to true if the request came through UDP, which will force standard-compliant clients to retry the request over TCP.

`policy.REROUTE`(*{subnet = target, ...}*)

Reroute IP 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. See *policy.add()* and do not forget to specify that this is *postrule*. Quick example:

```
-- this policy is enforced on answers
-- therefore we have to use 'postrule'
-- (the "true" at the end of policy.add)
policy.add(policy.all(policy.REROUTE({['192.0.2.0/24'] = '127.0.0.0'})), true)
```

`policy.ANSWER`(*{ type = { rdata=data, [ttl=1] } }, [nodata=false]*)

Overwrite Resource Records in responses with specified values.

- type - RR type to be replaced, e.g. `[kres.type.A]` or numeric value.
- rdata - RR data in DNS wire format, i.e. binary form specific for given RR type. Set of multiple RRs can be specified as table `{ rdata1, rdata2, ... }`. Use helper function `kres.str2ip()` to generate wire format for A and AAAA records. Wire format for other record types can be generated with *kres.parse_rdata()*.
- ttl - TTL in seconds. Default: 1 second.
- nodata - If type requested by client is not configured in this policy:
  - `true`: Return empty answer (*NODATA*).
  - `false`: Ignore this policy and continue processing other rules.

  Default: `false`.

```
-- policy to change IPv4 address and TTL for example.com
policy.add(
    policy.domains(
        policy.ANSWER(
            { [kres.type.A] = { rdata=kres.str2ip('192.0.2.7'), ttl=300 } }
        ), { todname('example.com') }))
-- policy to generate two TXT records (specified in binary format) for example.net
policy.add(
    policy.domains(
        policy.ANSWER(
            { [kres.type.TXT] = { rdata={'\005first', '\006second'}, ttl=5 } }
        ), { todname('example.net') }))
```

`kres.parse_rdata`(*{str, ...}*)

Parse string representation of RTYPE and RDATA into RDATA wire format. Expects a table of string(s) and returns a table of wire data.

```
-- create wire format RDATA that can be passed to policy.ANSWER
kres.parse_rdata({'SVCB 1 resolver.example. alpn=dot'})
kres.parse_rdata({
   'SVCB 1 resolver.example. alpn=dot ipv4hint=192.0.2.1 ipv6hint=2001:db8::1',
   'SVCB 2 resolver.example. mandatory=key65380 alpn=h2 key65380=/dns-query{?
```

(continues on next page)

(continued from previous page)

```
↪dns}',
})
```

More complex non-chain actions are described in their own chapters, namely:

- *Forwarding*
- *Response Policy Zones*

Chain actions

Following actions act on request and then processing continue until first non-chain action (specified in the previous section) is triggered:

`policy.`**`MIRROR`**`(`*ip_address*`)`

Send copy of incoming DNS queries to a given IP address using DNS-over-UDP and continue resolving them as usual. This is useful for sanity testing new versions of DNS resolvers.

```
policy.add(policy.all(policy.MIRROR('127.0.0.2')))
```

`policy.`**`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. Flag names correspond to *kr_qflags* structure. Use only if you know what you are doing.

Actions for extra logging

These are also "chain" actions, i.e. they don't stop processing the policy rule list. Similarly to other actions, they apply during whole processing of the client's request, i.e. including any sub-queries.

The log lines from these policy actions are tagged by extra `[reqdbg]` prefix, and they are produced regardless of your *log_level()* setting. They are marked as `debug` level, so e.g. with journalctl command you can use `-p info` to skip them.

Warning: Beware of producing too much logs.

These actions are not suitable for use on a large fraction of resolver's requests. The extra logs have significant performance impact and might also overload your logging system (or get rate-limited by it). You can use *Filters* to further limit on which requests this happens.

`policy.`**`DEBUG_ALWAYS`**

Print debug-level logging for this request. That also includes messages from client (*REQTRACE*), upstream servers (*QTRACE*), and stats about interesting records at the end.

```
-- debug requests that ask for flaky.example.net or below
policy.add(policy.suffix(policy.DEBUG_ALWAYS,
    policy.todnames({'flaky.example.net'})))
```

`policy.`**`DEBUG_CACHE_MISS`**

Same as *DEBUG_ALWAYS* but only if the request required information which was not available locally, i.e. requests which forced resolver to ask upstream server(s). Intended usage is for debugging problems with remote servers.

policy.**DEBUG_IF**(*test_function*)

Parameters

test_function – Function with single argument of type *kr_request* which returns `true` if debug logs for that request should be generated and `false` otherwise.

Same as *DEBUG_ALWAYS* but only logs if the test_function says so.

Note: `test_function` is evaluated only when request is finished. As a result all debug logs this request must be collected, and at the end they get either printed or thrown away.

Example usage which gathers verbose logs for all requests in subtree `dnssec-failed.org.` and prints debug logs for those finishing in a different state than `kres.DONE` (most importantly `kres.FAIL`, see *kr_layer_state*).

```
policy.add(policy.suffix(
    policy.DEBUG_IF(function(req)
                        return (req.state ~= kres.DONE)
                    end),
    policy.todnames({'dnssec-failed.org.'})))
```

policy.**QTRACE**

Pretty-print DNS responses from upstream servers (or cache) into logs. It's useful for debugging weird DNS servers.

If you do not use `QTRACE` in combination with `DEBUG*`, you additionally need either `log_groups({'iterat'})` (possibly with other groups) or `log_level('debug')` to see the output in logs.

policy.**REQTRACE**

Pretty-print DNS requests from clients into the verbose log. It's useful for debugging weird DNS clients. It makes most sense together with *Views and ACLs* (enabling per-client) and probably with verbose logging those request (e.g. use *DEBUG_ALWAYS* instead).

policy.**IPTRACE**

Log how the request arrived. Most notably, this includes the client's IP address, so beware of privacy implications.

```
-- example usage in configuration
policy.add(policy.all(policy.IPTRACE))
-- you might want to combine it with some other logs, e.g.
policy.add(policy.all(policy.DEBUG_ALWAYS))
```

```
-- example log lines from IPTRACE:
[reqdbg][policy][57517.00] request packet arrived from ::1#37931 to ::1#00853 (TCP␣
↪+ TLS)
[reqdbg][policy][65538.00] request packet arrived internally
```

Custom actions

policy.**custom_action**(*state*, *request*)

Parameters

- **state** – Request processing state *kr_layer_state*.
- **request** – Current DNS request as *kr_request* structure.

Returns

Returning a new *kr_layer_state* prevents evaluating other policy rules. Returning `nil` creates a *chain action* and allows to continue evaluating other rules.

This is real example of an action function:

```
-- Custom action which generates fake A record
local ffi = require('ffi')
local function fake_A_record(state, req)
    local answer = req:ensure_answer()
    if answer == nil then return nil end
    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
```

This custom action can be used as any other built-in action. For example this applies our *fake A record action* and executes it on all queries in subtree `example.net`:

```
policy.add(policy.suffix(fake_A_record, policy.todnames({'example.net'})))
```

The action function can implement arbitrary logic so 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.

Forwarding

Forwarding action alters behavior for cache-miss events. If an information is missing in the local cache the resolver will *forward* the query to *another DNS resolver* for resolution (instead of contacting authoritative servers directly). DNS answers from the remote resolver are then processed locally and sent back to the original client.

Actions *policy.FORWARD()*, *policy.TLS_FORWARD()* and *policy.STUB()* accept up to four IP addresses at once and the resolver will automatically select IP address which statistically responds the fastest.

policy.**FORWARD**(*ip_address*)

policy.**FORWARD**(*{ ip_address, [ip_address, ...] }*)

Forward cache-miss queries to specified IP addresses (without encryption), DNSSEC validate received answers and cache them. Target IP addresses are expected to be DNS resolvers.

```
-- Forward all queries to public resolvers https://www.nic.cz/odvr
policy.add(policy.all(
   policy.FORWARD(
       {'2001:148f:fffe::1', '2001:148f:ffff::1',
        '185.43.135.1', '193.14.47.1'})))
```

A variant which uses encrypted DNS-over-TLS transport is called *policy.TLS_FORWARD()*, please see section *Forwarding over TLS protocol (DNS-over-TLS)*.

policy.**STUB**(*ip_address*)

policy.**STUB**(*{ ip_address, [ip_address, ...] }*)

Similar to *policy.FORWARD()* 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.

This mode does not support encryption and should be used only for *Replacing part of the DNS tree*. Use *policy.FORWARD()* mode if possible.

```
-- Answers for reverse queries about the 192.168.1.0/24 subnet
-- are to be obtained from IP address 192.0.2.1 port 5353
-- This disables DNSSEC validation!
policy.add(policy.suffix(
    policy.STUB('192.0.2.1@5353'),
    {todname('1.168.192.in-addr.arpa')}))
```

Note: By default, forwarding targets must support EDNS and 0x20 randomization. See example in *Replacing part of the DNS tree*.

Warning: Limiting forwarding actions by filters (e.g. *policy.suffix()*) may have unexpected consequences. Notably, forwarders can inject *any* records into your cache even if you "restrict" them to an insignificant DNS subtree – except in cases where DNSSEC validation applies, of course.

The behavior is probably best understood through the fact that filters and actions are completely decoupled. The forwarding actions have no clue about why they were executed, e.g. that the user wanted to restrict the forwarder only to some subtree. The action just selects some set of forwarders to process this whole request from the client, and during that processing it might need some other "sub-queries" (e.g. for validation). Some of those might not've passed the intended filter, but policy rule-set only applies once per client's request.

Forwarding over TLS protocol (DNS-over-TLS)

policy.**TLS_FORWARD**(*{ {ip_address, authentication}, [...] }*)

Same as *policy.FORWARD()* but send query over DNS-over-TLS protocol (encrypted). Each target IP address needs explicit configuration how to validate TLS certificate so each IP address is configured by pair: `{ip_address, authentication}`. See sections below for more details.

Policy *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 *policy.TLS_FORWARD()* 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 conjunction with *policy.TLS_FORWARD()*, it's possible to forward different queries to different targets.

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 query name.

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 algorithm when the slicing function is initialized. By default, the assignment 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
```

These two functions can be used together to forward queries for names in different parts of DNS name space to different target servers:

```
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'},
```

(continues on next page)

(continued from previous page)

```
        {'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.

Replacing part of the DNS tree

Following procedure applies only to domains which have different content publicly and internally. For example this applies to "your own" top-level domain `example.` which does not exist in the public (global) DNS namespace.

Dealing with these internal-only domains requires extra configuration because DNS was designed as "single namespace" and local modifications like adding your own TLD break this assumption.

Warning: Use of internal names which are not delegated from the public DNS *is causing technical problems* with caching and DNSSEC validation and generally makes DNS operation more costly. We recommend **against** using these non-delegated names.

To make such internal domain available in your resolver it is necessary to *graft* your domain onto the public DNS namespace, but *grafting* creates new issues:

These *grafted* domains will be rejected by DNSSEC validation because such domains are technically indistinguishable from an spoofing attack against the public DNS. Therefore, if you trust the remote resolver which hosts the internal-only domain, and you trust your link to it, you need to use the *policy.STUB()* policy instead of *policy.FORWARD()* to disable DNSSEC validation for those *grafted* domains.

Listing 1: Example configuration grafting domains onto public DNS namespace

```
extraTrees = policy.todnames(
    {'faketldtest.',
     'sld.example.',
     'internal.example.com.',
     '2.0.192.in-addr.arpa.'  -- this applies to reverse DNS tree as well
     })
-- Beware: the rule order is important, as policy.STUB is not a chain action.
-- Flags: for "dumb" targets disabling EDNS can help (below) as DNSSEC isn't
-- validated anyway; in some of those cases adding 'NO_0X20' can also help,
-- though it also lowers defenses against off-path attacks on communication
-- between the two servers.
-- With kresd <= 5.5.3 you also needed 'NO_CACHE' flag to avoid unintentional
-- NXDOMAINs that could sometimes happen due to aggressive DNSSEC caching.
policy.add(policy.suffix(policy.FLAGS({'NO_EDNS'}), extraTrees))
policy.add(policy.suffix(policy.STUB({'2001:db8::1'}), extraTrees))
```

Response policy zones

Warning: There is no published Internet Standard for RPZ and implementations vary. At the moment Knot Resolver supports limited subset of RPZ format and deviates from implementation in BIND. Nevertheless it is good enough for blocking large lists of spam or advertising domains.

The RPZ file format is basically a DNS zone file with *very special* semantics. For example:

```
; left hand side          ; TTL and class  ; right hand side
; encodes RPZ trigger     ; ignored        ; encodes action
; (i.e. filter)
blocked.domain.example    600 IN           CNAME .          ; block main␣
↪domain
*.blocked.domain.example  600 IN           CNAME .          ; block subdomains
```

The only "trigger" supported in Knot Resolver is query name, i.e. left hand side must be a domain name which triggers the action specified on the right hand side.

Subset of possible RPZ actions is supported, namely:

RPZ Right Hand Side	Knot Resolver Action	BIND Compatibility
`.`	`action` is used	compatible if `action` is *`policy.DENY`*
`*.`	*`policy.ANSWER()`*	yes
`rpz-passthru.`	*`policy.PASS`*	yes
`rpz-tcp-only.`	*`policy.TC`*	yes
`rpz-drop.`	*`policy.DROP`*	no[1]
fake A/AAAA	*`policy.ANSWER()`*	yes
fake CNAME	not supported	no

Note: To debug which domains are affected by RPZ (or other policy actions), you can enable the `policy` log group:

```
log_groups({'policy'})
```

See also *non-ASCII support note*.

`policy.`**`rpz`**(*action*, *path*[, *watch* = *true*])

Parameters

- **`action`** – the default action for match in the zone; typically you want *`policy.DENY`*
- **`path`** – path to zone file
- **`watch`** – boolean, if true, the file will be 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.

For example, we can store the example snippet with domain `blocked.domain.example` (above) into file `/etc/knot-resolver/blocklist.rpz` and configure resolver to answer with *NXDOMAIN* plus the specified additional text to queries for this domain:

[1] Our *`policy.DROP`* returns *SERVFAIL* answer (for historical reasons).

```
policy.add(
    policy.rpz(policy.DENY_MSG('domain blocked by your resolver operator'),
               '/etc/knot-resolver/blocklist.rpz',
               true))
```

Resolver will reload RPZ file at run-time if the RPZ file changes. Recommended RPZ update procedure is to store new blocklist in a new file (*newblocklist.rpz*) and then rename the new file to the original file name (*blocklist.rpz*). This avoids problems where resolver might attempt to re-read an incomplete file.

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.

```
-- mirror all queries, keep handle so we can retrieve information later
local rule = policy.add(policy.all(policy.MIRROR('127.0.0.2')))
-- we can print statistics about this rule any time later
print(string.format('id: %d, matched queries: %d', rule.id, rule.count)
```

`policy.`**`del`**(*id*)

Parameters
`id` – identifier of a given rule returned by *policy.add()*

Returns
boolean `true` if rule was deleted, `false` otherwise

Remove a rule from policy list.

`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' }
```

7.6.2 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.

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 all IPv4 that hasn't matched
view:addr('0.0.0.0/0', policy.all(policy.DROP))
```

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>)
```

Properties

`view:addr(subnet, rule)`

Parameters

- **subnet** – client subnet, e.g. `10.0.0.1`
- **rule** – added rule, e.g. `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, e.g. `\5mykey`
- **rule** – added rule, e.g. `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.

7.6.3 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.

Tip: For blocking large lists of domains please use *policy.rpz()* instead of creating huge list of domains with IP address *0.0.0.0*.

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, so your hints might get surprisingly shadowed by even default policies.

That most often happens for **RFC 6761#section-6** names, e.g. `localhost` and `test` or with `PTR` records in private address ranges. To unblock the required names, you may use an explicit *policy.PASS* action.

```
policy.add(policy.suffix(policy.PASS, {todname('1.168.192.in-addr.arpa')}))
```

This `.PASS` workaround isn't ideal. To improve some cases, we recommend to move these `.PASS` lines to the end of your rule list. The point is that applying any *non-chain action* (e.g. *forwarding actions* or `.PASS` itself) stops processing *any* later policy rules for that request (including the default block-rules). You probably don't want this `.PASS` to shadow any other rules you might have; and on the other hand, if any other non-chain rule triggers, additional `.PASS` would not change anything even if it were somehow force-executed.

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.

7.6.4 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.

Simple example

```
-- Load the module with default settings
modules = { 'dns64' }
-- Reconfigure later
dns64.config({ prefix = '2001:db8::aabb:0:0' })
```

Warning: The module currently won't work well with *policy.STUB()*. Also, the IPv6 `prefix` passed in configuration is assumed to be `/96`.

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.

Advanced options

TTL in CNAME generated in the reverse `ip6.arpa.` subtree is configurable:

```
dns64.config({ prefix = '2001:db8:77ff::', rev_ttl = 300 })
```

You can specify a set of IPv6 subnets that are disallowed in answer. If they appear, they will be replaced by AAAAs generated from As.

```
dns64.config({
    prefix = '2001:db8:3::',
    exclude_subnets = { '2001:db8:888::/48', '::ffff/96' },
})
-- You could even pass '::/0' to always force using generated AAAAs.
```

In case you don't want dns64 for all clients, you can set `DNS64_DISABLE` flag via the *view module*.

```
modules = { 'dns64', 'view' }
-- disable dns64 for all IPv4 source addresses
view:addr('0.0.0.0/0', policy.all(policy.FLAGS('DNS64_DISABLE')))
-- disable dns64 for all IPv6 source addresses
view:addr('::/0', policy.all(policy.FLAGS('DNS64_DISABLE')))
-- re-enable dns64 for two IPv6 subnets
view:addr('2001:db8:11::/48', policy.all(policy.FLAGS(nil, 'DNS64_DISABLE')))
view:addr('2001:db8:93::/48', policy.all(policy.FLAGS(nil, 'DNS64_DISABLE')))
```

7.6.5 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.

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'},
                -- Remap /26 subnet (64 ip addresses)
                {'166.55.77.128/26', '127.0.0.192'},
                -- Remap a /32 block to a single address
                {'2001:db8::/32', '::1!'},
        }
}
```

7.6.6 Answer reordering

Certain clients are "dumb" and always connect to first IP address or name found in a DNS answer received from resolver instead 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 enabled by default since 5.3.0.

7.6.7 Rebinding protection

This module provides protection from DNS Rebinding attack by blocking answers which contain 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.

7.6.8 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')
```

7.6.9 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.

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 192.0.2.1 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')
```

(continues on next page)

(continued from previous page)

```
-- 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)

-- Delete all rules and start from scratch
daf.clear()
```

Warning: Only the first matching rule's action is executed. Defining additional actions for the same matching rule, e.g. `src = 127.0.0.1/8`, will have no effect.

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
}
```

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*.

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
}

# 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
```

7.7 Logging, monitoring, diagnostics

To read service logs use commands usual for your distribution. E.g. on distributions using systemd-journald use command `journalctl -u kresd@* -f`.

Knot Resolver supports 6 logging levels - `crit`, `err`, `warning`, `notice`, `info`, `debug`. All levels with the same meaning as is defined in `syslog.h`. It is possible change logging level using *log_level()* function.

```
log_level('debug') -- too verbose for normal usage
```

Logging level `notice` is set after start by default, so logs from Knot Resolver should contain only couple lines a day. For debugging purposes it is possible to use the very verbose `debug` level, but that is generally not usable unless restricted in some way (see below).

In addition to levels, logging is also divided into the *groups*. All groups are logged by default, but you can enable `debug` level for selected groups using *log_groups()* function. Other groups are logged to the log level set by *log_level()*.

It is also possible to enable `debug` logging level for particular requests, with *policies* or as *an HTTP service*.

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

`log_level`([*level*])

Param
string `'crit'`, `'err'`, `'warning'`, `'notice'`, `'info'` or `'debug'`

Returns
string Current logging level.

Pass a string to set the global logging level.

`verbose`([*true* | *false*])

Deprecated since version 5.4.0: Use *log_level()* instead.

Param
`true` enable `debug` level, `false` switch to default level (`notice`).

Returns
boolean `true` when `debug` level is enabled.

Toggle between `debug` and `notice` log level. Use only for debugging purposes. On busy systems verbose logging can produce several MB of logs per second and will slow down operation.

`log_target`(*target*)

Param
string `'syslog'`, `'stderr'`, `'stdout'`

Returns

string Current logging target.

Knot Resolver logs to standard error stream by default, but typical systemd units change that to `'syslog'`. That setting logs directly through systemd's facilities (if available) to preserve more meta-data.

`log_groups`([*table*])

Param
table of string(s) representing *log groups*

Returns
table of string with currently set log groups

Use to turn-on debug logging for the selected groups regardless of the global log level. Calling with no argument lists the currently active log groups. To remove all log groups, call the function with an empty table.

```
log_groups({'io', 'tls'}  -- turn on debug logging for io and tls groups
log_groups()              -- list active log groups
log_groups({})            -- remove all log groups
```

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*.

7.7.1 DNSSEC validation failure logging

This module logs a message for each DNSSEC validation failure (on `notice` *level*). 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()
{
    {
        ['count'] = 1,
        ['name'] = 'dnssec-failed.org.',
        ['type'] = 'DNSKEY',
    },
    {
        ['count'] = 13,
        ['name'] = 'rhybar.cz.',
        ['type'] = 'DNSKEY',
    },
}
```

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

7.7.2 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*.

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**)
request.xdp	external requests received over plain UDP via an AF_XDP socket

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.sum_ms	sum of all latencies in ms

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
> 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
}
```

(continues on next page)

(continued from previous page)

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

Module reference

stats.**get**(*key*)

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

Returns
number

Return nominal value of given metric.

stats.**set**(*'key val'*)

Set nominal value of given metric.

Example:

```
stats.set('answer.total 5')
-- or syntactic sugar
stats['answer.total'] = 5
```

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 overridden 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.

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() .. worker.id, -- 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 you 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.

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
```

(continues on next page)

(continued from previous page)

```
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
```

7.7.3 Scripting worker

Worker is a service over event loop that tracks and schedules outstanding queries, you can see the statistics or schedule new queries. It also contains information about specified worker count and process rank.

worker.id

Value from environment variable `SYSTEMD_INSTANCE`, or if it is not set, *PID* (string).

worker.pid

Current worker process PID (number).

worker.**stats**()

Return table of statistics. See member descriptions in *worker_stats*. A few fields are added, mainly from POSIX `getrusage()`:

- `usertime` and `systime` – CPU time used, in seconds
- `pagefaults` – the number of hard page faults, i.e. those that required I/O activity
- `swaps` – the number of times the process was "swapped" out of main memory; unused on Linux
- `csw` – the number of context switches, both voluntary and involuntary
- `rss` – current memory usage in bytes, including whole cache (resident set size)

Example:

```
print(worker.stats().concurrent)
```

7.7.4 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')
```

7.7.5 Debugging a single request

Using query policies

Query policies *policy.DEBUG_ALWAYS*, *policy.DEBUG_CACHE_MISS* or *policy.DEBUG_IF* can be used to enable debug-level logging for selected subdomains or queries matching specific conditions. Please refer to *Actions for extra logging* for more information.

Using HTTP module

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 debug-level 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
```

(continues on next page)

(continued from previous page)

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

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

7.7.6 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.

7.7.7 Dnstap (traffic collection)

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

The unix socket and the socket reader must be present before starting resolver instances. Also it needs appropriate filesystem permissions; the typical user and group of the daemon are called `knot-resolver`.

Tunables:

- `socket_path`: the unix socket file where dnstap messages will be sent
- `identity`: identity string as typically returned by an "NSID" (RFC 5001) query, empty by default
- `version`: version string of the resolver, defaulting to "Knot Resolver major.minor.patch"
- `client.log_queries`: if `true` queries from downstream in wire format will be logged
- `client.log_responses`: if `true` responses to downstream in wire format will be logged

```
modules = {
    dnstap = {
        socket_path = "/tmp/dnstap.sock",
        identity = nsid.name() or "",
        version = "My Custom Knot Resolver " .. package_version(),
        client = {
            log_queries = true,
            log_responses = true,
        },
    }
}
```

7.7.8 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.

7.7.9 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.

7.7.10 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.

7.7.11 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.

7.7.12 Debugging options

In case the resolver crashes, it is often helpful to collect a coredump from the crashed process. Configuring the system to collect coredump from crashed process is out of the scope of this documentation, but some tips can be found here.

Kresd uses its own mechanism for assertions. They are checks that should always pass and indicate some weird or unexpected state if they don't. In such cases, they show up in the log as errors. By default, the process recovers from those states if possible, but the behaviour can be changed with the following options to aid further debugging.

`debugging.assertion_abort = false|true`

> **Return**
> boolean (default: false in meson's release mode, true otherwise)

Allow the process to be aborted in case it encounters a failed assertion. (Some critical conditions always lead to abortion, regardless of settings.)

`debugging.assertion_fork = milliseconds`

> **Return**
> int (default: 5 minutes in meson's release mode, 0 otherwise)

If a process should be aborted, it can be done in two ways. When this is set to nonzero (default), a child is forked and aborted to obtain a coredump, while the parent process recovers and keeps running. This can be useful to debug a rare issue that occurs in production, since it doesn't affect the main process.

As the dumping can be costly, the value is a lower bound on delay between consecutive coredumps of each process. It is randomized by +-25% each time.

7.7.13 Logging API

Group names

LOG_GRP_SYSTEM_TAG

`system`: catch-all log for generic messages

LOG_GRP_CACHE_TAG

`cache`: operations related to cache

LOG_GRP_IO_TAG

`io`: input/output operations

LOG_GRP_NETWORK_TAG

`net`: network configuration and operation

LOG_GRP_TA_TAG

`ta`: basic log for trust anchors (TA)

LOG_GRP_TASENTINEL_TAG

`tasent`: TA sentinel

LOG_GRP_TASIGNALING_TAG

`tasign`: TA signal query

LOG_GRP_TAUPDATE_TAG

`taupd`: TA update

LOG_GRP_TLS_TAG

`tls`: TLS encryption layer

LOG_GRP_GNUTLS_TAG

`gnutls`: low-level logs from GnuTLS

LOG_GRP_TLSCLIENT_TAG

`tls_cl`: TLS client messages (used for TLS forwarding)

LOG_GRP_XDP_TAG

`xdp`: operations related to XDP

LOG_GRP_DOH_TAG

`doh`: DNS-over-HTTPS logger (doh2 implementation)

LOG_GRP_DNSSEC_TAG

`dnssec`: operations related to DNSSEC

LOG_GRP_HINT_TAG

`hint`: operations related to static hints

LOG_GRP_PLAN_TAG

`plan`: operations related to resolution plan

LOG_GRP_ITERATOR_TAG

`iterat`: operations related to iterate layer

LOG_GRP_VALIDATOR_TAG

`valdtr`: operations related to validate layer

LOG_GRP_RESOLVER_TAG

`resolv`: operations related to resolving

LOG_GRP_SELECTION_TAG

`select`: operations related to server selection

LOG_GRP_ZCUT_TAG

`zonecut`: operations related to zone cut

LOG_GRP_COOKIES_TAG

`cookie`: operations related to cookies

LOG_GRP_STATISTICS_TAG

`statis`: operations related to statistics

LOG_GRP_REBIND_TAG

`rebind`: operations related to rebinding

LOG_GRP_WORKER_TAG

`worker`: operations related to worker layer

LOG_GRP_POLICY_TAG

`policy`: operations related to policy

LOG_GRP_DAF_TAG

`daf`: operations related to DAF module

LOG_GRP_DETECTTIMEJUMP_TAG

`timejm`: operations related to time jump

LOG_GRP_DETECTTIMESKEW_TAG

`timesk`: operations related to time skew

LOG_GRP_GRAPHITE_TAG

`graphi`: operations related to graphite

LOG_GRP_PREFILL_TAG

`prefil`: operations related to prefill

LOG_GRP_PRIMING_TAG

`primin`: operations related to priming

LOG_GRP_SRVSTALE_TAG

`srvstl`: operations related to serve stale

LOG_GRP_WATCHDOG_TAG

`wtchdg`: operations related to watchdog

LOG_GRP_NSID_TAG

`nsid`: operations related to NSID

LOG_GRP_DNSTAP_TAG

`dnstap`: operations related to dnstap

LOG_GRP_TESTS_TAG

`tests`: operations related to tests

LOG_GRP_DOTAUTH_TAG

`dotaut`: DNS-over-TLS against authoritative servers

LOG_GRP_HTTP_TAG

`http`: http module, its web interface and legacy DNS-over-HTTPS

LOG_GRP_CONTROL_TAG

`contrl`: TTY control sockets

LOG_GRP_MODULE_TAG

`module`: suitable for user-defined modules

LOG_GRP_DEVEL_TAG

`devel`: for development purposes

LOG_GRP_RENUMBER_TAG

`renum`: operation related to renumber

LOG_GRP_EDE_TAG
: `exterr`: extended error module

LOG_GRP_REQDBG_TAG
: `reqdbg`: debug logs enabled by policy actions

Logging levels

We stick very close to POSIX syslog.h

kr_log_debug(grp, fmt, ...)
: Debugging message.

  Can be very verbose. The level is most often used through VERBOSE_MSG.

kr_log_info(grp, fmt, ...)

kr_log_notice(grp, fmt, ...)

LOG_DEFAULT_LEVEL
: Levels less severe than `notice` are not logged by default.

kr_log_warning(grp, fmt, ...)

kr_log_error(grp, fmt, ...)
: Significant error.

  The process continues, except for configuration errors during startup.

kr_log_crit(grp, fmt, ...)
: Critical condition.

  The process dies. Bad configuration should not cause this.

kr_log_deprecate(grp, fmt, ...)

kr_log(fmt, ...)
: Logging function for user modules.

  Uses group LOG_GRP_MODULE and `info` level.

  Parameters

  - **fmt** – Format string

Defines

LOG_UNKNOWN_LEVEL
: Negative error value.

LOG_GNUTLS_LEVEL
: GnuTLS level is 5.

KR_LOG_LEVEL_IS(exp)

kr_log_req(req, qry_id, indent, grp, fmt, ...)

Log a debug-level message from a *kr_request*.

Typically we call kr_log_q() instead.

Parameters

- **qry_uid** – query ID to append to request ID, 0 means "no query"
- **indent** – level of indentation between [group][req.qry] and message
- **grp** – GROUP_NAME (without the LOG_GRP_ prefix)
- **fmt** – printf-like format string

kr_log_q(qry, grp, fmt, ...)

Log a debug-level message from a *kr_query*.

Parameters

- **qry** – current query
- **grp** – GROUP_NAME (without the LOG_GRP_ prefix)
- **fmt** – printf-like format string

kr_log_is_debug(grp, req)

Return whether a particular log group in a request is in debug/verbose mode.

Typically you use this as condition to compute some data to be logged, in case that's considered too expensive to do unless it really gets logged.

The request can be NULL, and there's a _qry() shorthand to specify query instead.

kr_log_is_debug_qry(grp, qry)

KR_LOG_SJM_STR(x)

SD_JOURNAL_METADATA

Typedefs

typedef int **kr_log_level_t**

Enums

enum **kr_log_target_t**

Values:

enumerator **LOG_TARGET_SYSLOG**

enumerator **LOG_TARGET_STDERR**

enumerator **LOG_TARGET_STDOUT**

enumerator **LOG_TARGET_DEFAULT**

enum **kr_log_group**

Values:

enumerator **LOG_GRP_UNKNOWN**

enumerator **LOG_GRP_SYSTEM**

enumerator **LOG_GRP_CACHE**

enumerator **LOG_GRP_IO**

enumerator **LOG_GRP_NETWORK**

enumerator **LOG_GRP_TA**

enumerator **LOG_GRP_TLS**

enumerator **LOG_GRP_GNUTLS**

enumerator **LOG_GRP_TLSCLIENT**

enumerator **LOG_GRP_XDP**

enumerator **LOG_GRP_DOH**

enumerator **LOG_GRP_DNSSEC**

enumerator **LOG_GRP_HINT**

enumerator **LOG_GRP_PLAN**

enumerator **LOG_GRP_ITERATOR**

enumerator **LOG_GRP_VALIDATOR**

enumerator **LOG_GRP_RESOLVER**

enumerator **LOG_GRP_SELECTION**

enumerator **LOG_GRP_ZCUT**

enumerator **LOG_GRP_COOKIES**

enumerator **LOG_GRP_STATISTICS**

enumerator **LOG_GRP_REBIND**

enumerator **LOG_GRP_WORKER**

enumerator **LOG_GRP_POLICY**

enumerator **LOG_GRP_TASENTINEL**

enumerator **LOG_GRP_TASIGNALING**

enumerator **LOG_GRP_TAUPDATE**

enumerator **LOG_GRP_DAF**

enumerator **LOG_GRP_DETECTTIMEJUMP**

enumerator **LOG_GRP_DETECTTIMESKEW**

enumerator **LOG_GRP_GRAPHITE**

enumerator **LOG_GRP_PREFILL**

enumerator **LOG_GRP_PRIMING**

enumerator **LOG_GRP_SRVSTALE**

enumerator **LOG_GRP_WATCHDOG**

enumerator **LOG_GRP_NSID**

enumerator **LOG_GRP_DNSTAP**

enumerator **LOG_GRP_TESTS**

enumerator **LOG_GRP_DOTAUTH**

enumerator **LOG_GRP_HTTP**

enumerator **LOG_GRP_CONTROL**

enumerator **LOG_GRP_MODULE**

enumerator **LOG_GRP_DEVEL**

enumerator **LOG_GRP_RENUMBER**

enumerator **LOG_GRP_EDE**

enumerator **LOG_GRP_REQDBG**

Functions

void **kr_log_target_set**(*kr_log_target_t* target)

Set the current logging target.

bool **kr_log_group_is_set**(enum *kr_log_group* group)

void **kr_log_group_add**(enum *kr_log_group* group)

void **kr_log_group_reset**()

const char ***kr_log_grp2name**(enum *kr_log_group* group)

enum *kr_log_group* **kr_log_name2grp**(const char *name)

void **kr_log_level_set**(*kr_log_level_t* level)

Set the current logging level.

const char ***kr_log_level2name**(*kr_log_level_t* level)

kr_log_level_t **kr_log_name2level**(const char *name)

Return negative on error.

void **kr_log_req1**(const struct *kr_request* *const req, uint32_t qry_uid, const unsigned int indent, enum *kr_log_group* group, const char *tag, const char *fmt, ...)

void **kr_log_q1**(const struct *kr_query* *qry, enum *kr_log_group* group, const char *tag, const char *fmt, ...)

bool **kr_log_is_debug_fun**(enum *kr_log_group* group, const struct *kr_request* *req)

void **kr_log_fmt**(enum *kr_log_group* group, *kr_log_level_t* level, const char *file, const char *line, const char *func, const char *fmt, ...)

Variables

kr_log_target_t **kr_log_target**

Current logging target.

Read only, please.

kr_log_level_t **kr_log_level**

Current logging level.

Read only, please.

7.8 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
```

(continues on next page)

(continued from previous page)

```
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 effectively 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

7.9 Experimental features

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

[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.

7.9.1 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 embedded 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.

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 the 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 *Upgrading to 6.0.0 from 5.x.x*) 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. `socat`:

```
$ 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, which gives you an immediate response on the outcome of your command. The commands and their output are also logged in `contrl` group, on `debug` level if successful or `warning` level if failed (see around *log_level()*).

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.

map(*lua_snippet*)

Executes the provided string as lua code on every running resolver instance and returns the results as a table.

Key `n` is always present in the returned table and specifies the total number of instances the command was executed on. The table also contains results from each instance accessible through keys 1 to `n` (inclusive). If any instance returns `nil`, it is not explicitly part of the table, but you can detect it by iterating through 1 to `n`.

```
> map('worker.id')  -- return an ID of every active instance
{
    '2',
    '1',
    ['n'] = 2,
}
> map('worker.id == "1" or nil')  -- example of `nil` return value
```

(continues on next page)

(continued from previous page)

```
{
    [2] = true,
    ['n'] = 2,
}
```

The order of instances isn't guaranteed or stable. When you need to identify the instances, you may use `kluautil.kr_table_pack()` function to return multiple values as a table. It uses similar semantics with `n` as described above to allow `nil` values.

```
> map('require("kluautil").kr_table_pack(worker.id, stats.get("answer.total"))')
{
    {
        '2',
        42,
        ['n'] = 2,
    },
    {
        '1',
        69,
        ['n'] = 2,
    },
    ['n'] = 2,
}
```

If the command fails on any instance, an error is returned and the execution is in an undefined state (the command might not have been executed on all instances). When using the `map()` function to execute any code that might fail, your code should be wrapped in pcall() to avoid this issue.

```
> map('require("kluautil").kr_table_pack(pcall(net.tls, "cert.pem", "key.pem"))')
{
    {
        true,  -- function succeeded
        true,  -- function return value(s)
        ['n'] = 2,
    },
    {
        false,  -- function failed
        'error occurred...',  -- the returned error message
        ['n'] = 2,
    },
    ['n'] = 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')

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}}
```

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:

```
local ffi = require('ffi')
modules.load('stats')

-- log statistics every second
local stat_id = event.recurrent(1 * second, function(evid)
     log_info(ffi.C.LOG_GRP_STATISTICS, 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 function Lua uses concept called closures. In the following example function `speed_monitor()` is a closure function, which provides persistent variable called `previous`.

```
local ffi = require('ffi')
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_warn(ffi.C.LOG_GRP_STATISTICS, '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
    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

Execute function immediately and then periodically after each `interval`.

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!')
```

(continues on next page)

(continued from previous page)

```
   -- Halve 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
```

(continues on next page)

(continued from previous page)

```
    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 coroutines continue execution now
        log(testname .. ': system time AFTER sleep' .. tostring(os.time())
end

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
```

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-resolver/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 --lua-version 5.1 install etcd --from=https://mah0x211.github.io/
rocks/
```

7.9.2 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!

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=`.

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
...
...
```

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.

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.

Dependencies

- lua-basexx available in LuaRocks

CHAPTER

EIGHT

SYSTEMD

In the default installation, Knot Resolver contains systemd integration and starting it on such system usually involves only one command.

```
systemctl enable --now knot-resolver.service
```

If you don't have systemd service file for Knot Resolver already installed in your system, you can create one manually with the folling content:

```
[Unit]
Description=Knot Resolver Manager
Documentation=man:knot-resolver.systemd(7)
Wants=network-online.target
After=network-online.target
Before=nss-lookup.target
Wants=nss-lookup.target

[Service]
Type=notify
TimeoutStartSec=10s
ExecStart=@bin_dir@/knot-resolver --config=@etc_dir@/config.yml
ExecReload=@bin_dir@/kresctl --socket @run_dir@/manager.sock reload
KillSignal=SIGINT
WorkingDirectory=@systemd_work_dir@
User=@user@
Group=@group@
CapabilityBoundingSet=CAP_NET_BIND_SERVICE CAP_SETPCAP
AmbientCapabilities=CAP_NET_BIND_SERVICE CAP_SETPCAP

[Install]
WantedBy=multi-user.target
```

Note: Replace words surrounded by `@` to some real values (i.e. `@user@` to a user you want Knot Resolver to run as).

CHAPTER

NINE

MANUAL

The Knot Resolver can be started with the command `knot-resolver`. You can provide an optional argument `--config path/to/config.yml` to load a different than default configuration file.

The resolver does not have any external runtime dependencies and it should be able to run in most environments. It should be possible to wrap it with any container technology.

9.1 Multiple instances on a single server

The only limitation for running multiple instances of Knot Resolver is that all instances must have a different runtime directory. There are however safeguards in place that should prevent accidental runtime directory conflicts.

It is possible to share cache between multiple instances, just make sure that all instances have the same cache config and there is only a single garbage collector running (disable it in all but one config file).

CHAPTER

TEN

DOCKER

Note: Before version 6, our Docker images were not meant to be used in production. This is no longer the case and with the introduction of `kres-manager`, Knot Resolver runs in containers without any issues.

An official Docker image can be found on Docker Hub. The image contains Knot Resolver as if it was installed from our official distro packages.

```
docker run --rm -ti -P docker.io/cznic/knot-resolver
```

The configuration file is located at `/etc/knot-resolver/config.yml` and the cache is at `/var/cache/knot-resolver`. We recommend configuring a persistent cache across container restarts.

Warning: While the container image contains normal installation of Knot Resolver and there shouldn't be any differences between running it natively and in a container, we (the developers) do not have any experience using the Docker image in production. Especially, beware of running the DNS resolver with a software defined network (i.e. in Kubernetes). There will likely be some performance penalties for doing so. We haven't done any measurements comparing different types of installations so we don't know the performance differences. If you have done some measurements yourself, please reach out to us and we will share it here with everyone else.

CHAPTER

ELEVEN

ADVANCED

Warning: This page is intended for experienced users only. If you follow these instructions, you are not protected from footguns elimited with the introduction of the `kres-manager`. However, if you want to continue using Knot Resolver the same as before the version `6.0.0` this is a chapter for you.

For new and less experienced users, we recommend using the newer approach starting in the *Getting Started* chapter.

11.1 Usage without the manager

There are a few downsides to using the Knot Resolver without the manager:.

- Configuration is a imperative Lua script and can't be properly validated without actually running it.
- `kresd` is single-threaded so you need to manage multiple processes manually.
- Restarts without downtime after configuration change are only your responsibility.

11.1.1 Startup

The older way to start Knot Resolver is to run single instance of its resolving daemon manualy using `kresd@` systemd integration. The daemon is single thread process.

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

Tip: For more information about `systemd` integration see `man kresd.systemd`.

11.1.2 Configuration

You can configure `kresd` by pasting your Lua code into `/etc/knot-resolver/kresd.conf` configuration script. The resolver's daemon is preconfigure to load this script when using `kresd@` systemd integration.

Note: 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.

The first thing you need to configure are the network interfaces to listen to.

The following example instructs the 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' })
```

Complete configurations files examples 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).

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

Warning: This page is intended for experienced users only. If you follow these instructions, you are not protected from footguns elimited with the introduction of the `kres-manager`. However, if you want to continue using Knot Resolver the same as before the version `6.0.0` this is a chapter for you.

For new and less experienced users, we recommend using the newer approach starting in the *Getting Started* chapter.

11.2 Usage without systemd and without manager

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.

`kresd` 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.

Warning: This page is intended for experienced users only. If you follow these instructions, you are not protected from footguns elimited with the introduction of the `kres-manager`. However, if you want to continue using Knot Resolver the same as before the version `6.0.0` this is a chapter for you.

For new and less experienced users, we recommend using the newer approach starting in the *Getting Started* chapter.

11.2.1 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).

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
```

Warning: This page is intended for experienced users only. If you follow these instructions, you are not protected from footguns elimited with the introduction of the `kres-manager`. However, if you want to continue using Knot Resolver the same as before the version `6.0.0` this is a chapter for you.

For new and less experienced users, we recommend using the newer approach starting in the *Getting Started* chapter.

11.2.2 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.

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 capabilities after the daemon successfully starts when running as a non-root user.

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
```

Running as root

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

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

CHAPTER

TWELVE

HTTP API

12.1 Management HTTP API

You can use HTTP API to dynamically change configuration of already running Knot Resolver. By default the API is configured as UNIX domain socket `manager.sock` located in the resolver's rundir (typically `/run/knot-resolver/`). This socket is used by `kresctl` utility in default.

The API setting can be changed only in `/etc/knot-resolver/config.yml` configuration file:

```
management:
    interface: 127.0.0.1@5000
    # or use unix socket instead of inteface
    # unix-socket: /my/new/socket.sock
```

First version of configuration API endpoint is available on `/v1/config` HTTP endpoint. Configuration API supports following HTTP request methods:

HTTP request methods	Operation
GET `/v1/config[/path]`	returns current configuration with an ETag
PUT `/v1/config[/path]`	upsert (try update, if does not exists, insert), appends to array
PATCH `/v1/config[/path]`	update property using JSON Patch
DELETE `/v1/config[/path]`	delete an existing property or list item at given index

Note: Managemnet API has other useful endpoints (metrics, schema, ...), see the detailed *API documentation*.

path:
Determines specific configuration option or configuration subtree on that path. Items in lists and dictionaries are reachable using indexes `/list-name/{index}/` and keys `/dict-name/{key}/`.

payload:
JSON or YAML encoding is used for configuration payload.

Note: Some configuration options cannot be configured via the API for stability and security reasons(e.g. API configuration itself). In the case of an attempt to configure such an option, the operation is rejected.

12.2 Dynamically changing configuration

Knot Resolver Manager is capable of dynamically changing its configuration via an HTTP API or by reloading its config file. Both methods are equivalent in terms of its capabilities. The `kresctl` utility uses the HTTP API and provides a convinient command line interface.

12.2.1 Reloading configuration file

To reload the configuration file, send the `SIGHUP` signal to the Manager process. The original configuration file will be read again, validated and in case of no errors, the changes will be applied.

Note: You can also send `SIGHUP` to the top-level process, to the supervisord. Normally, supervisord would stop all processes and reload its configuration when it receives SIGHUP. However, we have eliminated this footgun in order to prevent anyone from accidentally shutting down the whole resolver. Instead, the signal is only forwarded to the Manager.

12.2.2 HTTP API

Listen address

By default, the Manager exposes its HTTP API on a Unix socket at `FIXME`. However, you can change where it listens by changing the `management.interface` config option. To use `kresctl`, you have to tell it this value.

List of API endpoints

- `GET /schema` returns JSON schema of the configuration data model
- `GET /schema/ui` redirect to an external website with the JSON schema visualization
- `GET /metrics` provides Prometheus metrics
- `GET /` static response that could be used to determine, whether the Manager is running
- `POST /stop` gracefully stops the Manager, empty request body
- `{GET,PUT,DELETE,PATCH} /v1/config` allows reading and modifying current configuration

Config modification endpoint (v1)

Note: The `v1` version qualifier is there for future-proofing. We don't have any plans at the moment to change the API any time soon. If that happens, we will support both old and new API versions for the some transition period.

The API by default expects JSON, but can also parse YAML when the `Content-Type` header is set to `application/yaml` or `text/vnd.yaml`. The return value is always a JSON with `Content-Type: application/json`. The schema of input and output is always a subtree of the configuration data model which is described by the JSON schema exposed at `/schema`.

The API can operate on any configuration subtree by specifying a JSON pointer in the URL path (property names and list indices joined with `/`). For example, to get the number of worker processes, you can send `GET` request to `v1/config/workers`.

The different HTTP methods perform different modifications of the configuration:

- `GET` return subtree of the current configuration

- `PUT` set property
- `DELETE` removes the given property or list item at the given index
- `PATCH` updates the configuration using JSON Patch

To prevent race conditions when changing configuration from multiple clients simultaneously, every response from the Manager has an `ETag` header set. Requests then accept `If-Match` and `If-None-Match` headers with the latest `ETag` value and the corresponding request processing fails with HTTP error code 412 (precondition failed).

CHAPTER

THIRTEEN

KRESCTL UTILITY

Command-line utility that helps communicate with the *management API*. It also provides tooling to work with declarative configuration (`validate`, `convert`).

-h, --help

Shows help message. It can be also used with every *command* for its help message.

13.1 Connecting to the management API

Most *commands* require connection to the *management API*. With default Knot Resolver configuration, `kresctl` should communicate with the resolver withou need to specify `--socket` option. If not, this option must be set for each command.

-s <socket>, **--socket** <socket>

Default
"./manager.sock"

Optional, path to Unix-domain socket or network interface of the *management API*.

```
$ kresctl --socket http://127.0.0.1@5000 {command} # network interface, port 5000
$ kresctl --socket /path/to/socket.sock {command}  # unix-domain socket location
```

13.2 Commands

The following possitional arguments determine what kind of command will be executed. Only one of these arguments can be selected during the execution of a single `krestctl` command.

config

Performs operations on the running resolver's configuration. Requires connection to the management API.

Operations:

Use one of the following operations to be performed on the configuration.

get

Get current configuration from the resolver.

-p <path>, **--path** <path>

Optional, path (JSON pointer, RFC6901) to the configuration resources. By default, the entire configuration is selected.

`--json, --yaml`

Default

--json

Get configuration data in JSON or YAML format.

`<file>`

Optional, path to the file where to save exported configuration data. If not specified, data will be printed.

`set`

Set new configuration for the resolver.

`-p` <path>, `--path` <path>

Optional, path (JSON pointer, RFC6901) to the configuration resources. By default, the entire configuration is selected.

`--json, --yaml`

Default

--json

Set configuration data in JSON or YAML format.

`[ <file> | <value> ]`

Optional, path to file with new configuraion or new configuration value. If not specified, value will be readed from stdin.

`delete`

Delete given configuration property or list item at the given index.

`-p` <path>, `--path` <path>

Optional, path (JSON pointer, RFC6901) to the configuration resources. By default, the entire configuration is selected.

This command reads current `network` configuration subtree from the resolver and exports it to file in YAML format.

```
$ kresctl config get --yaml -p /network ./network-config.yaml
```

Next command changes workers configuration to `8`.

```
$ kresctl config set -p /workers 8
```

`metrics`

Reads agregated metrics data in Propmetheus format directly from the running resolver. Requires connection to the management API.

`<file>`

Optional, file where to export Prometheus metrics. If not specified, the metrics are printed.

```
$ kresctl metrics ./metrics/data.txt
```

`schema`

Shows JSON-schema repersentation of the Knot Resolver's configuration.

`-l, --live`

Get configuration JSON-schema from the running resolver. Requires connection to the management API.

<file>

Optional, file where to export JSON-schema. If not specified, the JSON-schema is printed.

```
$ kresctl schema --live ./mydir/config-schema.json
```

validate

Validates configuration in JSON or YAML format.

<input_file>

File with configuration in YAML or JSON format.

```
$ kresctl validate input-config.json
```

convert

Converts JSON or YAML configuration to Lua script.

<input_file>

File with configuration in YAML or JSON format.

<output_file>

Optional, output file for converted configuration in Lua script. If not specified, converted configuration is printed.

```
$ kresctl convert input-config.yaml output-script.lua
```

reload

Tells the resolver to reload YAML configuration file. Old processes are replaced by new ones (with updated configuration) using rolling restarts. So there will be no DNS service unavailability during reload operation. Requires connection to the management API.

stop

Tells the resolver to shutdown everthing. No process will run after this command. Requires connection to the management API.

CHAPTER

FOURTEEN

UPGRADING TO 6.0.0 FROM 5.X.X

Version 6 of Knot Resolver brings one significant change - it introduces *Knot Resolver Manager* - a new way for interacting with Knot Resolver. The Manager brings several new features:

- **new declarative configuration**
- HTTP API to change configuration on the fly without downtime
- it hides complexities of running multiple instances of `kresd`

Now, you might be worried about the future of `kresd`. No worries, you can use `kresd` directly the same way you did before, nothing changes there right now. However, in the long run, we might make breaking changes in the way `kresd` is configured and using it directly is from now on considered advanced.

With the release of version 6, there is a new way to configure and control your running `kresd` instances so that you don't have to configure multiple systemd services. The new Knot Resolver Manager handles it for you. In the table below, you can find comparison of how things were done before and how they can be done now.

14.1 Command rosetta

In the table below, you can compare the way Knot Resolver was used before and how it can be used now.

Task	How to do it now	How it was done before
start resolver	`systemctl start knot-resolver`	`systemctl start kresd@1`
stop resolver	`systemctl stop knot-resolver`	`systemctl stop kresd@1`
start resolver with 4 worker processes	set `/workers` to 4 in the config file	manually start 4 services by `systemctl start kresd@{1,2,3,4}`
rolling restart after updating config	`systemctl reload knot-resolver` (or use API or `kresctl`)	manually restart individual `kresd@` services one by one
open logs of all instances	`journalctl -u knot-resolver`	`journalctl -u system-kresd.slice`
open log of a single kresd instances	`journalctl -u knot-resolver _PID=xxx`	`journalctl -u kresd@1`
updating config programatically	use HTTP API or `kresctl` command	write a custom tool to generate new config and restart `kresd`'s
handling errors during config changes	HTTP API just reports error, resolver keeps running with previous config	custom tools for every user
validate new config	`kresctl validate path/to/new/config.yml` (not fully bullet proof), then try to run it	run `kresd` with the config and see if it fails
look at the Lua config	`kresctl convert path/to/new/config.yml`	`cat /path/to/config.conf`
gather metrics	point Prometheus etc. at the single HTTP API	collect metrics manually from all individual processes

CHAPTER

FIFTEEN

UPGRADING

This section summarizes steps required when upgrading to newer Knot Resolver versions. We advise users to also read *Release notes* for respective versions. Section *Module changes* is relevant only for users who develop or use third-party modules.

15.1 Upcoming changes

Following section provides information about selected changes in not-yet-released versions. We advise users to prepare for these changes sooner rather than later to make it easier to upgrade to newer versions when they are released.

- Command line option `--forks` (`-f`) is deprecated and will be eventually removed. Preferred way to manage *Multiple instances* is to use a process manager, e.g. systemd or supervisord.
- Function `verbose()` is deprecated and will be eventually removed. Prefered way to change logging level is use to `log_level()`.

15.2 5.x to 6.0

- *detailed upgrade guide <upgrading-to-6>*

15.3 5.4 to 5.5

15.3.1 Packagers & Developers

- Knot DNS >= 3.0.2 is required.

15.3.2 Module API changes

- Function *cache.zone_import* was removed; you can use *ffi.C.zi_zone_import* instead (different API).
- When using *PROXYv2 protocol*, the meaning of `qsource.flags` and `qsource.comm_flags` in `kr_request` changes so that `flags` describes the original client communicating with the proxy, while `comm_flags` describes the proxy communicating with the resolver. When there is no proxy, `flags` and `comm_flags` are the same.

15.4 5.3 to 5.4

15.4.1 Configuration file

- `kind='doh'` in *net.listen()* was renamed to `kind='doh_legacy'`. It is recommended to switch to the new DoH implementation with `kind='doh2'`.
- *verbose()* has been deprecated. In case you want to change logging level, there is new function *log_level()*.

15.4.2 Packagers & Developers

- meson option `verbose_log` was removed.

15.4.3 Module changes

- lua function `warn()` was removed, use `log_warn()` instead. The new function takes a log group number as the first argument.
- C functions `kr_log_req()` and `kr_log_q()` were replaced by `kr_log_req1()` and `kr_log_q1()` respectively. The new function have slightly different API.

15.5 5.2 to 5.3

15.5.1 Configuration file

- Module `dnstap`: option `log_responses` has been moved inside a new `client` section. Refer to the configuration example in *Dnstap (traffic collection)*.

15.5.2 Packagers & Developers

- Knot DNS >= 2.9 is required.

15.6 5.1 to 5.2

15.6.1 Users

- DoH over HTTP/1 and unencrypted transports is still available in *legacy http module* (`kind='doh'`). This module will not receive receive any more bugfixes and will be eventually removed.
- Users of *Control sockets* API need to terminate each command sent to resolver with newline character (ASCII `\n`). Correct usage: `cache.stats()\n`. Newline terminated commands are accepted by all resolver versions >= 1.0.0.
- DNS Flag Day 2020 is now effective and Knot Resolver uses maximum size of UDP answer to 1232 bytes. Please double-check your firewall, it has to allow DNS traffic on UDP and **also TCP** port 53.

- Human readable output in interactive mode and from *Control sockets* was improved and as consequence slightly changed its format. Users who need machine readable output for scripts should use Lua function `tojson()` to convert Lua values into standard JSON format instead of attempting to parse the human readable output. For example API call `tojson(cache.stats())\n` will return JSON string with `cache.stats()` results represented as dictionary. Function `tojson()` is available in all resolver versions >= 1.0.0.

15.6.2 Configuration file

- Statistics exporter *Graphite/InfluxDB/Metronome* now uses default prefix which combines *`hostname()`* and *`worker.id`* instead of bare *`hostname()`*. This prevents *Multiple instances* from sending conflicting statistics to server. In case you want to continue in previous time series you can manually set the old values using option `prefix` in *Graphite configuration*. Beware that non-default values require careful *Instance-specific configuration* to avoid conflicting names.
- Lua variable *`worker.id`* is now a string with either Systemd instance name or PID (instead of number). If your custom configuration uses *`worker.id`* value please check your scripts.

15.6.3 Module changes

- Reply packet *`kr_request.answer`* is not allocated immediately when the request comes. See the new *`kr_request_ensure_answer()`* function, wrapped for lua as `req:ensure_answer()`.

15.7 5.0 to 5.1

15.7.1 Module changes

- Modules which use *`kr_request.trace_log`* handler need update to modified handler API. Example migration is modules/watchdog/watchdog.lua.
- Modules which were using logger `kr_log_qverbose_impl()` need migration to new logger *`kr_log_q()`*. Example migration is modules/rebinding/rebinding.lua.
- Modules which were using *`kr_ranked_rrarray_add()`* should note that on success it no longer returns exclusively zero but index into the array (non-negative). Error states are unchanged (negative).

15.8 4.x to 5.x

15.8.1 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).

15.8.2 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.

15.9 4.2.2 to 4.3+

15.9.1 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.

15.10 4.x to 4.2.1+

15.10.1 Users

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

15.11 3.x to 4.x

15.11.1 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 *Legacy DNS-over-HTTPS (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 endpoints, 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*

15.11.2 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.

15.12 2.x to 3.x

15.12.1 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.)

15.12.2 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

SIXTEEN

RELEASE NOTES

16.1 Version numbering

Version number format is `major.minor.patch`. Knot Resolver does not use semantic versioning even though the version number looks similar.

Leftmost number which was changed signalizes what to expect when upgrading:

Major version

- Manual upgrade steps might be necessary, please follow instructions in *Upgrading* section.
- Major releases may contain significant changes including changes to configuration format.
- We might release a new major also when internal implementation details change significantly.

Minor version

- Configuration stays compatible with the previous version, except for undocumented or very obscure options.
- Upgrade should be seamless for users who use modules shipped as part of Knot Resolver distribution.
- Incompatible changes in internal APIs are allowed in minor versions. Users who develop or use custom modules (i.e. modules not distributed together with Knot Resolver) need to double check their modules for incompatibilities. *Upgrading* section should contain hints for module authors.

Patch version

- Everything should be compatible with the previous version.
- API for modules should be stable on best effort basis, i.e. API is very unlikely to break in patch releases.
- Custom modules might need to be recompiled, i.e. ABI compatibility is not guaranteed.

This definition is not applicable to versions older than 5.2.0.

16.2 Knot Resolver 6.0.0 (2023-mm-dd)

16.2.1 Improvements

- Knot Resolver v6 alpha starts
- 6.0.x versions are dedicated to alpha cycle

16.3 Knot Resolver 5.6.0 (2023-01-26)

16.3.1 Security

- avoid excessive TCP reconnections in some cases (!1380) For example, a DNS server that just closes connections without answer could cause lots of work for the resolver (and itself, too). The number of connections could be up to around 100 per client's query.

  We thank Xiang Li from NISL Lab, Tsinghua University, and Xuesong Bai and Qifan Zhang from DSP Lab, UCI.

16.3.2 Improvements

- daemon: feed server selection with more kinds of bad-answer events (!1380)
- cache.max_ttl(): lower the default from six days to one day and apply both limits to the first uncached answer already (!1323 #127)
- depend on jemalloc, preferably, to improve memory usage (!1353)
- no longer accept DNS messages with trailing data (!1365)
- policy.STUB: avoid applying aggressive DNSSEC denial proofs (!1364)
- policy.STUB: avoid copying +dnssec flag from client to upstream (!1364)

16.3.3 Bugfixes

- policy.DEBUG_IF: don't print client's packet unconditionally (!1366)

16.4 Knot Resolver 5.5.3 (2022-09-21)

16.4.1 Security

- fix CPU-expensive DoS by malicious domains - CVE-2022-40188

16.4.2 Improvements

- fix config_tests on macOS (both HW variants)

16.5 Knot Resolver 5.5.2 (2022-08-16)

16.5.1 Improvements

- support libknot 3.2 (!1309)
- priming module: hide failures from the default log level (!1310)
- reduce memory usage in some cases (!1328)

16.5.2 Bugfixes

- daemon/http: improve URI checks to fix some proxies (#746, !1311)
- daemon/tls: fix a double-free for some cases of policy.TLS_FORWARD (!1314)
- hints module: improve parsing comments in hosts files (!1315)
- renumber module: fix renumbering with name matching again (#760, !1334)

16.6 Knot Resolver 5.5.1 (2022-06-14)

16.6.1 Improvements

- daemon/tls: disable TLS resumption via tickets for TLS <= 1.2 (#742, !1295)
- daemon/http: DoH now responds with proper HTTP codes (#728, !1279)
- renumber module: allow rewriting subnet to a single IP (!1302)
- renumber module: allow arbitrary netmask (!1306)
- nameserver selection algorithm: improve IPv6 avoidance if broken (!1298)

16.6.2 Bugfixes

- modules/dns64: fix incorrect packet writes for cached packets (#727, !1275)
- xdp: make it work also with libknot 3.1 (#735, !1276)
- prefill module: fix lockup when starting multiple idle instances (!1285)
- validator: fix some failing negative NSEC proofs (!1294, #738, #443)

16.7 Knot Resolver 5.5.0 (2022-03-15)

16.7.1 Improvements

- extended_errors: module for extended DNS error support, RFC8914 (!1234)
- policy: log policy actions; useful for RPZ debugging (!1239)
- policy: new action policy.IPTRACE for logging request origin (!1239)
- prefill module: prepare for ZONEMD, improve performance (!1225)
- validator: conditionally ignore SHA1 DS, as SHOULD by RFC4509 (!1251)
- lib/resolve: use EDNS padding for outgoing TLS queries (!1254)
- support for PROXYv2 protocol (!1238)
- lib/resolve, policy: new NO_ANSWER flag for not responding to clients (!1257)

16.7.2 Incompatible changes

- libknot >= 3.0.2 is required

16.7.3 Bugfixes

- doh2: fix CORS by adding *access-control-allow-origin:* * (!1246)
- net: fix listen by interface - add interface suffix to link-local IPv6 (!1253)
- daemon/tls: fix resumption for outgoing TLS (e.g. TLS_FORWARD) (!1261)
- nameserver selection: fix interaction of timeouts with reboots (#722, !1269)

16.8 Knot Resolver 5.4.4 (2022-01-05)

16.8.1 Bugfixes

- fix bad zone cut update in certain cases (e.g. AWS; !1237)

16.9 Knot Resolver 5.4.3 (2021-12-01)

16.9.1 Improvements

- lua: add kres.parse_rdata() to parse RDATA from string to wire format (!1233)
- lua: add policy.domains() for exact domain name matching (!1228)

16.9.2 Bugfixes

- policy.rpz: fix origin detection in files without $ORIGIN (!1215)
- lua: log() works again; broken in 5.4.2 (!1223)
- policy: correctly include EDNS0 previously omitted by some actions (!1230)
- edns_keepalive: module is now properly loaded (!1229, thanks Josh Soref!)

16.10 Knot Resolver 5.4.2 (2021-10-13)

16.10.1 Improvements

- dns64 module: also map the reverse (PTR) subtree (#478, !1201)
- dns64 module: allow disabling based on client address (#368, !1201)
- dns64 module: allow configuring AAAA subnets not allowed in answer (!1201)
- nameserver selection algorithm: improve IPv6 avoidance if broken (!1207)

16.10.2 Bugfixes

- lua: log() output is visible with default log level again (!1208)
- build: fix when knot-dns headers are on non-standard location (!1210)

16.11 Knot Resolver 5.4.1 (2021-08-19)

16.11.1 Improvements

- docker: base image on Debian 11 (!1203)

16.11.2 Bugfixes

- fix build without doh2 support after 5.4.0 (!1197)
- fix policy.DEBUG* logging and -V/–version after 5.4.0 (!1199)
- doh2: ensure memory from unsent streams is freed (!1202)

16.12 Knot Resolver 5.4.0 (2021-07-29)

16.12.1 Improvements

- fine grained logging and syslog support (!1181)
- expose HTTP headers for processing DoH requests (!1165)
- improve assertion mechanism for debugging (!1146)
- support apkg tool for packaging workflow (!1178)
- support Knot DNS 3.1 (!1192, !1194)

16.12.2 Bugfixes

- trust_anchors.set_insecure: improve precision (#673, !1177)
- plug memory leaks related to TCP (!1182)
- policy.FLAGS: fix not applying properly in edge cases (!1179)
- fix a crash with older libuv inside timer processing (!1195)

16.12.3 Incompatible changes

- see upgrading guide: https://knot-resolver.readthedocs.io/en/stable/upgrading.html#to-5-4
- legacy DoH implementation configuration in net.listen() was renamed from kind="doh" to kind="doh_legacy" (!1180)

16.13 Knot Resolver 5.3.2 (2021-05-05)

16.13.1 Security

- validator: fix 5.3.1 regression on over-limit NSEC3 edge case (!1169) Assertion might be triggered by query/answer, potentially DoS. CVE-2021-40083 was later assigned.

16.13.2 Improvements

- cache: improve handling write errors from LMDB (!1159)
- doh2: improve handling of stream errors (!1164)

16.13.3 Bugfixes

- dnstap module: fix repeated configuration (!1168)
- validator: fix SERVFAIL for some rare dynamic proofs (!1166)
- fix SIGBUS on uncommon ARM machines (unaligned access; !1167, #426)
- cache: better resilience on abnormal termination/restarts (!1172)
- doh2: fix memleak on stream write failures (!1161)

16.14 Knot Resolver 5.3.1 (2021-03-31)

16.14.1 Improvements

- policy.STUB: try to avoid TCP (compared to 5.3.0; !1155)
- validator: downgrade NSEC3 records with too many iterations (>150; !1160)
- additional improvements to nameserver selection algorithm (!1154, !1150)

16.14.2 Bugfixes

- dnstap module: don't break request resolution on dnstap errors (!1147)
- cache garbage collector: fix crashes introduced in 5.3.0 (!1153)
- policy.TLS_FORWARD: better avoid dead addresses (#671, !1156)

16.15 Knot Resolver 5.3.0 (2021-02-25)

16.15.1 Improvements

- more consistency in using parent-side records for NS addresses (!1097)
- better algorithm for choosing nameservers (!1030, !1126, !1140, !1141, !1143)
- daf module: add daf.clear() (!1114)
- dnstap module: more features and don't log internal requests (!1103)
- dnstap module: include in upstream packages and Docker image (!1110, !1118)
- randomize record order by default, i.e. reorder_RR(true) (!1124)
- prometheus module: transform graphite tags into prometheus labels (!1109)
- avoid excessive logging of UDP replies with sendmmsg (!1138)

16.15.2 Bugfixes

- view: fail config if bad subnet is specified (!1112)
- doh2: fix memory leak (!1117)
- policy.ANSWER: minor fixes, mainly around NODATA answers (!1129)
- http, watchdog modules: fix stability problems (!1136)

16.15.3 Incompatible changes

- dnstap module: *log_responses* option gets nested under *client*; see new docs for config example (!1103)
- libknot >= 2.9 is required

16.16 Knot Resolver 5.2.1 (2020-12-09)

16.16.1 Improvements

- doh2: send Cache-Control header with TTL (#617, !1095)

16.16.2 Bugfixes

- fix map() command on 32-bit platforms; regressed in 5.2.0 (!1093)
- doh2: restrict endpoints to doh and dns-query (#636, !1104)
- renumber: map to correct subnet when using multiple rules (!1107)

16.17 Knot Resolver 5.2.0 (2020-11-11)

16.17.1 Improvements

- doh2: add native C module for DNS-over-HTTPS (#600, !997)
- xdp: add server-side XDP support for higher UDP performance (#533, !1083)
- lower default EDNS buffer size to 1232 bytes (#538, #300, !920); see https://www.dnsflagday.net/2020/
- net: split the EDNS buffer size into upstream and downstream (!1026)
- lua-http doh: answer to /dns-query endpoint as well as /doh (!1069)
- improve resiliency against UDP fragmentation attacks (disable PMTUD) (!1061)
- ta_update: warn if there are differences between statically configured keys and upstream (#251, !1051)
- human readable output in interactive mode was improved
- doc: generate info page (!1079)
- packaging: improve sysusers and tmpfiles support (!1080)

16.17.2 Bugfixes

- avoid an assert() error in stash_rrset() (!1072)
- fix emergency cache locking bug introduced in 5.1.3 (!1078)
- migrate map() command to control sockets; fix systemd integration (!1000)
- fix crash when sending back errors over control socket (!1000)
- fix SERVFAIL while processing forwarded CNAME to a sibling zone (#614, !1070)

16.17.3 Incompatible changes

- see upgrading guide: https://knot-resolver.readthedocs.io/en/stable/upgrading.html#to-5-2
- minor changes in module API
- control socket API commands have to be terminated by "n"
- graphite: default prefix now contains instance identifier (!1000)
- build: meson >= 0.49 is required (!1082)

16.18 Knot Resolver 5.1.3 (2020-09-08)

16.18.1 Improvements

- capabilities are no longer constrained when running as root (!1012)
- cache: add percentage usage to cache.stats() (#580, !1025)
- cache: add number of cache entries to cache.stats() (#510, !1028)
- aarch64 support again, as some systems still didn't work (!1033)
- support building against Knot DNS 3.0 (!1053)

16.18.2 Bugfixes

- tls: fix compilation to support net.tls_sticket_secret() (!1021)
- validator: ignore bogus RRSIGs present in insecure domains (!1022, #587)
- build if libsystemd version isn't detected as integer (#592, !1029)
- validator: more robust reaction on missing RRSIGs (#390, !1020)
- ta_update module: fix broken RFC5011 rollover (!1035)
- garbage collector: avoid keeping multiple copies of cache (!1042)

16.19 Knot Resolver 5.1.2 (2020-07-01)

16.19.1 Bugfixes

- hints module: NODATA answers also for non-address queries (!1005)
- tls: send alert to peer if handshake fails (!1007)
- cache: fix interaction between LMDB locks and preallocation (!1013)
- cache garbage collector: fix flushing of messages to logs (!1009)
- cache garbage collector: fix insufficient GC on 32-bit systems (!1009)
- graphite module: do not block resolver on TCP failures (!1014)
- policy.rpz various fixes (!1016): $ORIGIN issues, precision of warnings, allow answering with multi-RR sets

16.20 Knot Resolver 5.1.1 (2020-05-19)

16.20.1 Security

- fix CVE-2020-12667: mitigation for NXNSAttack DNS protocol vulnerability

16.20.2 Bugfixes

- control sockets: recognize newline as command boundary

16.21 Knot Resolver 5.1.0 (2020-04-29)

16.21.1 Improvements

- cache garbage collector: reduce filesystem operations when idle (!946)
- policy.DEBUG_ALWAYS and policy.DEBUG_IF for limited verbose logging (!957)
- daemon: improve TCP query latency under heavy TCP load (!968)
- add policy.ANSWER action (!964, #192)
- policy.rpz support fake A/AAAA (!964, #194)

16.21.2 Bugfixes

- cache: missing filesystem support for pre-allocation is no longer fatal (#549)
- lua: policy.rpz() no longer watches the file when watch is set to false (!954)
- fix a strict aliasing problem that might've lead to "miscompilation" (!962)
- fix handling of DNAMEs, especially signed ones (#234, !965)
- lua resolve(): correctly include EDNS0 in the virtual packet (!963) Custom modules might have been confused by that.
- do not leak bogus data into SERVFAIL answers (#396)
- improve random Lua number generator initialization (!979)
- cache: fix CNAME caching when validation is disabled (#472, !974)
- cache: fix CNAME caching in policy.STUB mode (!974)
- prefill: fix crash caused by race condition with resolver startup (!983)
- webmgmt: use javascript scheme detection for websockets' protocol (#546)
- daf module: fix del(), deny(), drop(), tc(), pass() functions (#553, !966)
- policy and daf modules: expose initial query when evaluating postrules (#556)
- cache: fix some cases of caching answers over 4 KiB (!976)
- docs: support sphinx 3.0.0+ (!978)

16.21.3 Incompatible changes

- minor changes in module API; see upgrading guide: https://knot-resolver.readthedocs.io/en/stable/upgrading.html

16.22 Knot Resolver 5.0.1 (2020-02-05)

16.22.1 Bugfixes

- systemd: use correct cache location for garbage collector (#543)

16.22.2 Improvements

- cache: add cache.fssize() lua function to configure entire free disk space on dedicated cache partition (#524, !932)

16.23 Knot Resolver 5.0.0 (2020-01-27)

16.23.1 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)

16.23.2 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)

16.23.3 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)

16.24 Knot Resolver 4.3.0 (2019-12-04)

16.24.1 Security - CVE-2019-19331

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

16.24.2 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)

16.24.3 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)

16.25 Knot Resolver 4.2.2 (2019-10-07)

16.25.1 Bugfixes

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

16.26 Knot Resolver 4.2.1 (2019-09-26)

16.26.1 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)

16.26.2 Improvements

- add compatibility with (future) libknot 2.9

16.27 Knot Resolver 4.2.0 (2019-08-05)

16.27.1 Improvements

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

16.27.2 Bugfixes

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

16.27.3 Module API changes

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

16.28 Knot Resolver 4.1.0 (2019-07-10)

16.28.1 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)

16.28.2 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)

16.28.3 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)

16.28.4 Module API changes

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

16.29 Knot Resolver 4.0.0 (2019-04-18)

16.29.1 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

16.29.2 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

16.29.3 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)

16.29.4 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.

16.30 Knot Resolver 3.2.1 (2019-01-10)

16.30.1 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!

16.30.2 Improvements

- improve handling of timed out 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

16.31 Knot Resolver 3.2.0 (2018-12-17)

16.31.1 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

16.31.2 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)

16.31.3 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)

16.31.4 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

16.32 Knot Resolver 3.1.0 (2018-11-02)

16.32.1 Incompatible changes

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

16.32.2 Improvements

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

16.32.3 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)

16.33 Knot Resolver 3.0.0 (2018-08-20)

16.33.1 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 about new releases please subscribe to https://lists.nic.cz/postorius/lists/knot-resolver-announce.lists.nic.cz/

16.33.2 Bugfixes

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

16.33.3 Improvements

- reorder_RR() implementation is brought back
- bring in performance 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

16.34 Knot Resolver 2.4.1 (2018-08-02)

16.34.1 Security

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

16.34.2 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)

16.35 Knot Resolver 2.4.0 (2018-07-03)

16.35.1 Incompatible changes

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

16.35.2 Security

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

16.35.3 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)

16.35.4 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)

16.35.5 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)

16.36 Knot Resolver 2.3.0 (2018-04-23)

16.36.1 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)

16.36.2 New features

- new policy.REFUSE to reply REFUSED to clients

16.36.3 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 unnecessary 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)

16.36.4 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)

16.37 Knot Resolver 2.2.0 (2018-03-28)

16.37.1 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.)

16.37.2 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

16.38 Knot Resolver 2.1.1 (2018-02-23)

16.38.1 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.

16.39 Knot Resolver 2.1.0 (2018-02-16)

16.39.1 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

16.39.2 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

16.40 Knot Resolver 2.0.0 (2018-01-31)

16.40.1 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)

16.40.2 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

16.40.3 Bugfixes

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

16.41 Knot Resolver 1.5.3 (2018-01-23)

16.41.1 Bugfixes

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

16.42 Knot Resolver 1.5.2 (2018-01-22)

16.42.1 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.

16.42.2 Bugfixes

- memcached: fix fallout from module rename in 1.5.1

16.43 Knot Resolver 1.5.1 (2017-12-12)

16.43.1 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

16.43.2 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)

16.43.3 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

16.44 Knot Resolver 1.5.0 (2017-11-02)

16.44.1 Bugfixes

- fix loading modules on Darwin

16.44.2 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)

16.45 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.

16.45.1 Improvements

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

16.45.2 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?

16.46 Knot Resolver 1.4.0 (2017-09-22)

16.46.1 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.

16.46.2 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).

16.46.3 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

16.47 Knot Resolver 1.3.3 (2017-08-09)

16.47.1 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.

16.47.2 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

16.47.3 Improvements

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

16.48 Knot Resolver 1.3.2 (2017-07-28)

16.48.1 Security

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

16.48.2 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)

16.48.3 Improvements

- embedded LMDB fallback: update 0.9.18 -> 0.9.21

16.49 Knot Resolver 1.3.1 (2017-06-23)

16.49.1 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

16.50 Knot Resolver 1.3.0 (2017-06-13)

16.50.1 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.

16.50.2 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.

16.50.3 Bugfixes

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

16.51 Knot Resolver 1.2.6 (2017-04-24)

16.51.1 Security

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

16.51.2 Improvements

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

16.51.3 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

16.52 Knot Resolver 1.2.5 (2017-04-05)

16.52.1 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

16.52.2 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)

16.52.3 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 timed out
- 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

16.53 Knot Resolver 1.2.4 (2017-03-09)

16.53.1 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.

16.53.2 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

16.53.3 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-existence 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

16.54 Knot Resolver 1.2.3 (2017-02-23)

16.54.1 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

16.55 Knot Resolver 1.2.2 (2017-02-10)

16.55.1 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

16.55.2 Testing:

- Update tests to run tests with and without QNAME minimization

16.56 Knot Resolver 1.2.1 (2017-02-01)

16.56.1 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.

16.56.2 Documentation

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

16.56.3 Bugfixes:

- lua: make the map command check its arguments

16.57 Knot Resolver 1.2.0 (2017-01-24)

16.57.1 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.

16.57.2 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}_ttl option, with max_ttl defaulting to 6 days.
- Option to reorder RRs in the response.
- New policy.QTRACE policy to print packet contents

16.57.3 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.

16.57.4 Miscellaneous:

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

16.58 Knot Resolver 1.1.1 (2016-08-24)

16.58.1 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

16.58.2 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

16.59 Knot Resolver 1.1.0 (2016-08-12)

16.59.1 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

16.60 Knot Resolver 1.0.0 (2016-05-30)

16.60.1 Initial release:

- The first initial release

CHAPTER

SEVENTEEN

SYSTEM ARCHITECTURE

As mentioned in the getting started section, Knot Resolver is split into several components, namely the manager, `kresd` and the garbage collector. In addition to these custom components, we also rely on supervisord.

There are two different control structures in place. Semantically, the manager controls every other component in Knot Resolver. It processes configuration and passes it onto every other component. As a user you will always interact with the manager (or kresd). At the same time though, the manager is not the root of the process hierarchy, Supervisord sits at the top of the process tree and runs everything else.

Note: The rationale for this inverted process hierarchy is mainly stability. Supervisord sits at the top because it is a reliable and stable software we can depend upon. It also does not process user input and its therefore shielded from data processing bugs. This way, any component in Knot Resolver can crash and restart without impacting the rest of the system.

17.1 Knot Resolver startup

The inverted process hierarchy complicates Resolver's launch procedure. You might notice it when reading manager's logs just after start. What happens on cold start is:

1. Manager starts, reads its configuration and generates new supervisord configuration. Then, it starts supervisord by using `exec`.
2. Supervisord loads it's configuration, loads our extensions and start a new instance of manager.
3. Manager starts again, this time as a child of supervisord. As this is desired state, it loads the configuration again and commands supervisord that it should start new instances of `kresd`.

17.2 Failure handling

Knot Resolver is designed to handle failures automatically. Anything except for supervisord will automatically restart. If a failure is irrecoverable, all processes will stop and nothing will be left behind in a half-broken state. While a total failure like this should never happen, it is possible and you should not rely on single instance of Knot Resolver for a highly-available system.

Note: The ability to restart most of the components without downtime means, that Knot Resolver is able to transparently apply updates while running.

17.3 Individual components

You can learn more about architecture of individual Resolver components in the following chapters.

17.3.1 kres-manager

Note: This guide is intended for advanced users and developers. You don't have to know and understand any of this to use Knot Resolver.

The manager is a component written in Python and a bit of C used for native extension modules. The main goal of the manager is to ensure the system is set up according to a given configuration, provide a user-friendly interface. Performance is only secondary to correctness.

The manager is mostly modelled around config processing pipeline:

API

The API server is implemented using aiohttp. This framework provides the application skeleton and manages application runtime. The manager is actually a normal web application with the slight difference that we don't save the data in a database but rather modify state of other processes.

Code of the API server is located only in a single source code file. It also contains description of the manager's startup procedure.

Config processing

From the web framework, we receive data as simple strings and we need to parse and validate them. Due to packaging issues in distros, we rolled our own solution not disimilar to Python library Pydantic.

Our tool lets us model config schema similarly to how Python's native dataclasses are constructed. As input, it takes Python's dicts taken from PyYAML or JSON parser. The dict is mapped onto predefined Python classes while enforcing typing rules. If desired, the mapping step is performed multiple times onto different classes, which allows us to process intermediary values such as `auto`.

There are two relevant places in the source code - our generic modelling tools and the actual configuration data model. Just next to the data model in the `templates` directory, there are Jinja2 templates for generating Lua code from the configuration.

Actual manager

The actual core of the whole application is originally named the manager. It keeps a high-level view of the systems state and performs all necessary operations to change the state to the desired one. In other words, manager is the component handling rolling restarts, config update logic and more.

The code is contained mainly in a single source code file.

Interactions with supervisord

Note: Let's make a sidestep and let's talk about abstractions. The manager component mentioned above interacts with a general backend (or as we call sometimes call it - a subprocess manager). The idea is that the interactions with the backend are not dependent on the backend's implementation and we can choose which one we want to use. Historically, we had two different backend implementations - systemd and supervisord. However, systemd turned out to be inappropriate, it did not fit our needs, so we removed it. The abstraction remains though and it should be possible to implement a different subprocess manager if it turns out useful. Please note though, the abstraction might be somewhat leaky in practice as there is only one implementation.

Communication with supervisord happens on pretty much all possible levels. We edit its configuration file, we use its XMLRPC API, we use Unix signals and we even attach to it from within its Python runtime. The interface is honestly a bit messy and we had to use all we could to make it user friendly.

First, we generate supervisord's configuration file. The configuration file sets stage for further communication by specifying location of the pidfile and API Unix socket. It prepares configuration for subprocesses and most significantly, it loads our custom extensions.

The extensions don't use a lot of code. There are four of them - the simplest one provides a speedier XMLRPC API for starting processes, it removes delays that are not necessary for our usecase. Another one implements systemd's `sd_notify()` API for supervisord, so we can track the lifecycle of ``kresd``s more precisely. Another extension changes the way logging works and the last extension monitors the lifecycle of the manager and forwards some signals.

Note: The extensions mentioned above use monkeypatching to achieve their design goals. We settled for this approach, because supervisord's codebase appears mostly stable. The code we patch has not been changed for years. Other option would be forking supervisord and vendoring it. We decided against that mainly due to packaging complications it would cause with major Linux distributions.

For executing subprocesses, we don't actually change the configuration file, we only use XMLRPC API and tell supervisord to start already configured programs. For one specific call though, we use our extension instead of the build-in method of starting processes as it is significantly faster.

17.3.2 kresd

17.3.3 kres-cache-gc

The garbage collector is a simple component which keeps the shared cache from overfilling. Every second it estimates cache usage and if over 80%, records get deleted in order to free 10%. (Parameters can be configured.)

The freeing happens in a few passes. First all items are classified by their estimated usefulness, in a simple way based on remaining TTL, type, etc. From this histogram it's computed which "level of usefulness" will become the threshold, so that roughly the planned total size gets freed. Then all items are passed to collect the set of keys to delete, and finally the deletion is performed. As longer transactions can cause issues in LMDB, all passes are split into short batches.

CHAPTER

EIGHTEEN

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.nic.cz/knot/knot-resolver.git
```

18.1 Building with apkg

Knot Resolver uses apkg tool for upstream packaging. It allows build packages localy for supported distributions, which it then installs. `apkg` also takes care of dependencies itself.

First, you need to install and setup `apkg`.

Tip: Install `apkg` with pipx to avoid version conflicts.

```
$ pip3 install apkg
$ apkg system-setup
```

Clone and change dir to `knot-resolver` git repository.

```
$ git clone --recursive https://gitlab.nic.cz/knot/knot-resolver.git
$ cd knot-resolver
```

Tip: The `apkg status` command can be used to find out some useful information, such as whether the current distribution is supported.

When `apkg` is ready, a package can be built and installed.

```
# takes care of dependencies
apkg build-dep

# build package
apkg build
```

(continues on next page)

(continued from previous page)

```
# (build and) install package, builds package when it is not already built
apkg install
```

After that Knot Resolver should be installed.

18.2 Building with Meson

Knot Resolver uses Meson Build system. Shell snippets below should be sufficient for basic usage but users unfamiliar with Meson might want to read introductory article Using Meson.

18.2.1 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 with core functions:

Requirement	Notes
ninja	*build only*
meson >= 0.49	*build only*[1]
C and C++ compiler	*build only*[2]
pkg-config	*build only*[3]
libknot 3.0.2+	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

Additional dependencies are needed to build and run Knot Resolver with `manager`: All dependencies are also listed in pyproject.toml which is our authoritative source.

Requirement	Notes
python3 >=3.6.8	Python language interpreter
Jinja2	Template engine for Python
PyYAML	YAML framework for Python
aiohttp	HTTP Client/Server for Python.
prometheus-client	Prometheus client for Python
typing-extensions	Compatibility module for Python

There are also *optional* packages that enable specific functionality in Knot Resolver:

[1] If `meson >= 0.49` 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`).

Optional	Needed for	Notes
jemalloc	`daemon`	Improve long-term memory consumption.
nghttp2	`daemon`	DNS over HTTPS support.
libsystemd	`daemon`	Systemd watchdog support.
libcap-ng	`daemon`	Linux capabilities: support dropping them.
lua-basexx	`config tests`	Number base encoding/decoding for Lua.
lua-http	`modules/http`	HTTP/2 client/server for Lua.
lua-cqueues	some lua modules	
cmocka	`unit tests`	Unit testing framework.
dnsdist	`proxyv2 test`	DNS proxy server
Doxygen	`documentation`	Generating API documentation.
Sphinx, sphinx-tabs and sphinx_rtd_theme	`documentation`	Building this documentation.
Texinfo	`documentation`	Generating this documentation in Info format.
breathe	`documentation`	Exposing Doxygen API doc to Sphinx.
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.

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.

18.2.2 Compilation

Folowing meson command creates new build directory named `build_dir`, configures installation path to `/tmp/kr` and enables static build (to allow installation to non-standard path). You can also configure some *Build options*, in this case enable `manager`, which is disabled by default.

```
$ meson build_dir --prefix=/tmp/kr --default-library=static -Dmanager=enabled
```

After that it is possible to build and install Knot Resolver.

```
$ meson setup build_dir --prefix=/tmp/kr --default-library=static
$ ninja -C build_dir
```

(continues on next page)

(continued from previous page)

```
# install Knot Resolver into the previously configured '/tmp/kr' path
$ 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.

18.2.3 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 setup build_dir
$ meson configure build_dir
```

To customize project build options, use `-Doption=value` when creating a build directory:

```
$ meson setup build_dir -Ddoc=enabled
```

... or change options in an already existing build directory:

```
$ meson configure build_dir -Ddoc=enabled
```

18.2.4 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.

18.3 Tests

The following is a non-comprehensitve lists of various tests that can be found in this repo. These can be enabled by the build system.

18.3.1 Unit tests

The unit tests depend on cmocka and can easily be executed after compilation. They are enabled by default (if `cmocka` is found).

```
$ ninja -C build_dir
$ meson test -C build_dir --suite unit
```

18.3.2 Postinstall tests

There following tests require a working installation of kresd. The binary `kresd` found in `$PATH` will be tested. When testing through meson, `$PATH` is modified automatically and you just need to make sure to install kresd first.

```
$ ninja install -C build_dir
```

18.3.3 Config tests

Config tests utilize the kresd's lua config file to execute arbitrary tests, typically testing various modules, their API etc.

To enable these tests, specify `-Dconfig_tests=enabled` option for meson. Multiple dependencies are required (refer to meson's output when configuring the build dir).

```
$ meson configure build_dir -Dconfig_tests=enabled
$ ninja install -C build_dir
$ meson test -C build_dir --suite config
```

18.3.4 Extra tests

The extra tests require a large set of additional dependencies and executing them outside of upstream development is probably redundant.

To enable these tests, specify `-Dextra_tests=enabled` option for meson. Multiple dependencies are required (refer to meson's output when configuring the build dir). Enabling `extra_tests` automatically enables config tests as well.

Integration tests

The integration tests are using Deckard, the DNS test harness. The tests simulate specific DNS scenarios, including authoritative server and their responses. These tests rely on linux namespaces, refer to Deckard documentation for more info.

```
$ meson configure build_dir -Dextra_tests=enabled
$ ninja install -C build_dir
$ meson test -C build_dir --suite integration
```

Pytests

The pytest suite is designed to spin up a kresd instance, acquire a connected socket, and then performs any tests on it. These tests are used to test for example TCP, TLS and its connection management.

```
$ meson configure build_dir -Dextra_tests=enabled
$ ninja install -C build_dir
$ meson test -C build_dir --suite pytests
```

18.3.5 Useful meson commands

It's 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
```

18.4 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 configure build_dir -Ddoc=enabled
$ ninja -C build_dir doc
```

18.5 Tarball

Released tarballs are available from https://knot-resolver.cz/download/

To make a release tarball from git, use the following command. The

```
$ ninja -C build_dir dist
```

It's also possible to make a development snapshot tarball:

```
$ ./scripts/make-archive.sh
```

18.6 Packaging

Recommended build options for packagers:

- `--buildtype=release` for default flags (optimization, 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 documentation (see *Documentation*)
- `-Dinstall_kresd_conf=enabled` to install default config file
- `-Dmanager=enabled` to force build of the manager and its features
- `-Dclient=enabled` to force build of kresc
- `-Dunit_tests=enabled` to force build of unit tests

18.6.1 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
```

18.6.2 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!).

18.7 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

NINETEEN

KNOT RESOLVER LIBRARY

19.1 Requirements

- libknot 2.0 (Knot DNS high-performance DNS library.)

19.2 For users

The library as described provides basic services for name resolution, which should cover the usage, examples are in the *resolve API* documentation.

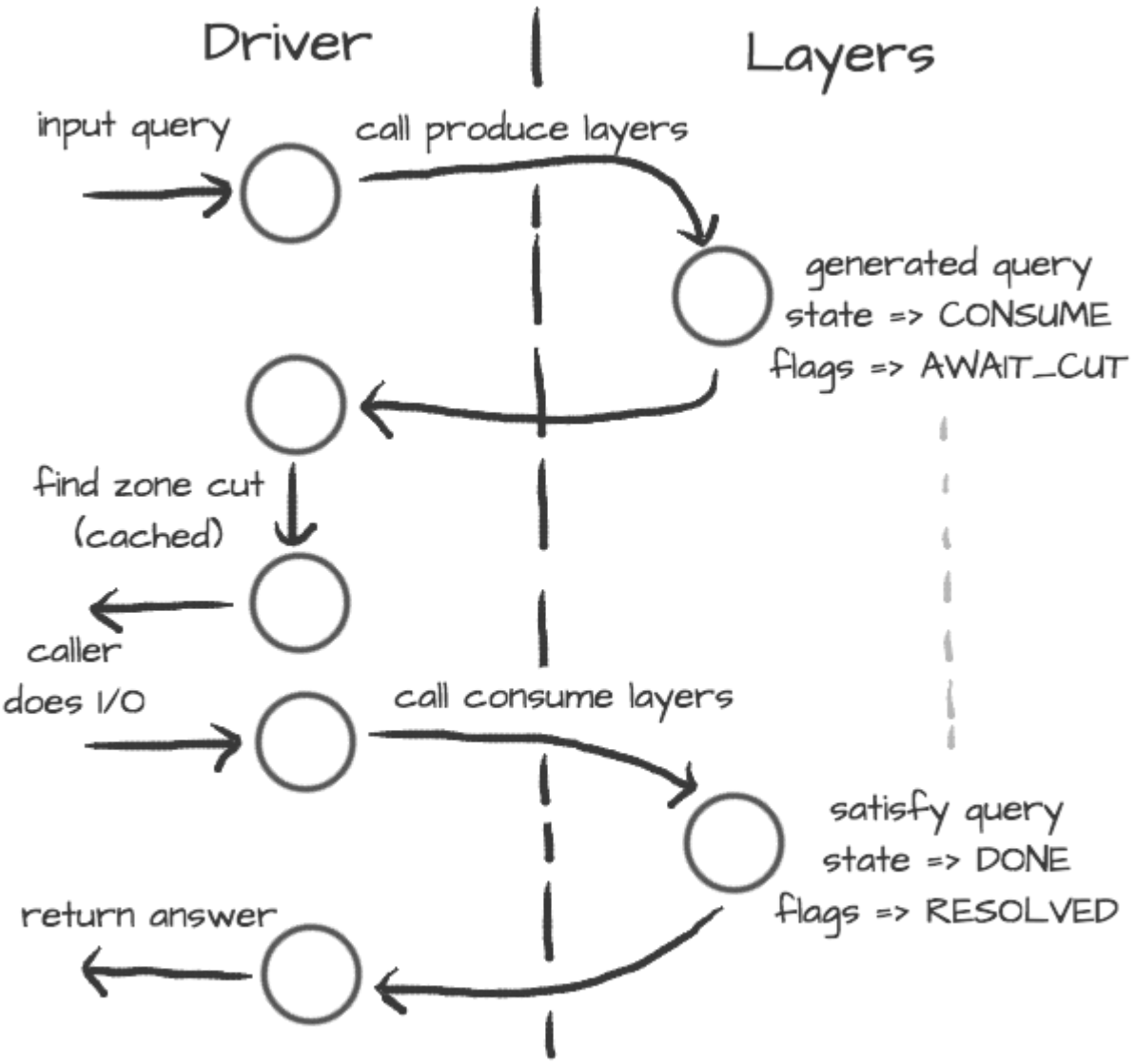
Tip: If you're migrating from `getaddrinfo()`, see *"synchronous"* API, but the library offers iterative API as well to plug it into your event loop for example.

19.3 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)
                qry.flags.AWAIT_CUT = true
```

(continues on next page)

(continued from previous page)

```
        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.

19.4 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;
}
```

19.5 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.

19.5.1 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`.
- Extended DNS error codes are in `kres.extended_error` table, e.g. `kres.extended_error.BLOCKED`.
- Packet sections (QUESTION, ANSWER, AUTHORITY, ADDITIONAL) are in the `kres.section` table.

19.5.2 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)
```

19.5.3 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))
```

19.5.4 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)
```

19.5.5 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()
```

(continues on next page)

(continued from previous page)

```
        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)
```

19.5.6 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.

19.6 API reference

Warning: This section is generated with doxygen and breathe. Due to their limitations, some symbols may be incorrectly described or missing entirely. For exhaustive and accurate reference, refer to the header files instead.

19.6.1 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);
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);
}
```

(continues on next page)

(continued from previous page)

```
// "state" is either DONE or FAIL
kr_resolve_finish(&request, state);
```

Defines

kr_request_selected(req)

Initializer for an array of *_selected.

Typedefs

typedef uint8_t *(***alloc_wire_f**)(struct *kr_request* *req, uint16_t *maxlen)

Allocate buffer for answer's wire (*maxlen may get lowered).

Motivation: XDP wire allocation is an overlap of library and daemon:

- it needs to be called from the library
- it needs to rely on some daemon's internals
- the library (currently) isn't allowed to directly use symbols from daemon (contrary to modules), e.g. some of our lib-using tests run without daemon

Note: after we obtain the wire, we're obliged to send it out. (So far there's no use case to allow cancelling at that point.)

typedef bool (***addr_info_f**)(struct sockaddr*)

typedef void (***async_resolution_f**)(knot_dname_t*, enum knot_rr_type)

typedef see_source_code **kr_sockaddr_array_t**

Enums

enum **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:

enumerator **KR_RANK_INITIAL**

Did not attempt to validate.

It's assumed compulsory to validate (or prove insecure).

enumerator **KR_RANK_OMIT**

Do not attempt to validate.

(And don't consider it a validation failure.)

enumerator **KR_RANK_TRY**

Attempt to validate, but failures are non-fatal.

enumerator **KR_RANK_INDET**

Unable to determine whether it should be secure.

enumerator **KR_RANK_BOGUS**

Ought to be secure but isn't.

enumerator **KR_RANK_MISMATCH**

enumerator **KR_RANK_MISSING**

No RRSIG found for that owner+type combination.

enumerator **KR_RANK_INSECURE**

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. Or it may be covered by a closer negative TA.

enumerator **KR_RANK_AUTH**

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).

enumerator **KR_RANK_SECURE**

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.

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 inline void **kr_rank_set**(uint8_t *rank, uint8_t kr_flag)

Set the rank state.

The _AUTH flag is kept as it was.

int **kr_resolve_begin**(struct *kr_request* *request, struct *kr_context* *ctx)

Begin name resolution.

Note: Expects a request to have an initialized mempool.

Parameters

- **request** – request state with initialized mempool
- **ctx** – resolution context

Returns

CONSUME (expecting query)

knot_rrset_t ***kr_request_ensure_edns**(struct *kr_request* *request)

Ensure that request->answer->opt_rr is present if query has EDNS.

This function should be used after clearing a response packet to ensure its opt_rr is properly set. Returns the opt_rr (for convenience) or NULL.

knot_pkt_t ***kr_request_ensure_answer**(struct *kr_request* *request)

Ensure that request->answer is usable, and return it (for convenience).

It may return NULL, in which case it marks ->state with _FAIL and no answer will be sent. Only use this when it's guaranteed that there will be no delay before sending it. You don't need to call this in places where "resolver knows" that there will be no delay, but even there you need to check if the ->answer is NULL (unless you check for _FAIL anyway).

int **kr_resolve_consume**(struct *kr_request* *request, struct *kr_transport* **transport, 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.

Parameters

- **request** – request state (awaiting input)
- **src** – [in] packet source address
- **packet** – [in] input packet

Returns
any state

int **kr_resolve_produce**(struct *kr_request* *request, struct *kr_transport* **transport, 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.

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

Returns
any state

int **kr_resolve_checkout**(struct *kr_request* *request, const struct sockaddr *src, struct *kr_transport* *transport, 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.

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

Returns
kr_ok() or error code

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.

Parameters

- **request** – request state
- **state** – either DONE or FAIL state (to be assigned to request->state)

Returns
DONE

struct *kr_rplan* ***kr_resolve_plan**(struct *kr_request* *request)

Return resolution plan.

Parameters

- **request** – request state

Returns
pointer to rplan

knot_mm_t ***kr_resolve_pool**(struct *kr_request* *request)

Return memory pool associated with request.

Parameters

- **request** – request state

Returns
mempool

int **kr_request_set_extended_error**(struct *kr_request* *request, int info_code, const char *extra_text)

Set the extended DNS error for request.

The error is set only if it has a higher or the same priority as the one already assigned. The provided extra_text may be NULL, or a string that is allocated either statically, or on the request's mempool. To clear any error, call it with KNOT_EDNS_EDE_NONE and NULL as extra_text.

To facilitate debugging, we include a unique base32 identifier at the start of the extra_text field for every call of this function. To generate such an identifier, you can use the command: $ base32 /dev/random | head -c 4

Parameters

- **request** – request state
- **info_code** – extended DNS error code
- **extra_text** – optional string with additional information

Returns
info_code that is set after the call

static inline void **kr_query_inform_timeout**(struct *kr_request* *req, const struct *kr_query* *qry)

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**

Default *kr_request* flags.

For startup defaults see init_resolver()

knot_rrset_t ***downstream_opt_rr**

Default EDNS towards *both* clients and upstream.

LATER: consider splitting the two, e.g. allow separately configured limits for UDP packet size (say, LAN is under control).

knot_rrset_t ***upstream_opt_rr**

trie_t ***trust_anchors**

trie_t ***negative_anchors**

struct *kr_zonecut* **root_hints**

struct *kr_cache* **cache**

unsigned **cache_rtt_tout_retry_interval**

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 **tcp**

true if the request is not on UDP; only meaningful if (dst_addr).

bool **tls**

true if the request is encrypted; only meaningful if (dst_addr).

bool **http**

true if the request is on HTTP; only meaningful if (dst_addr).

bool **xdp**

true if the request is on AF_XDP; only meaningful if (dst_addr).

struct **kr_extended_error**

Public Members

int32_t **info_code**

May contain -1 (KNOT_EDNS_EDE_NONE); filter before converting to uint16_t.

const char ***extra_text**

Can be NULL.

Allocated on the kr_request::pool or static.

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**

See *kr_request_ensure_answer()*

struct *kr_query* ***current_query**

Current evaluated query.

const struct sockaddr ***addr**

Address that originated the request.

May be that of a client behind a proxy, if PROXYv2 is used. Otherwise, it will be the same as `comm_addr`. `NULL` for internal origin.

const struct sockaddr ***comm_addr**

Address that communicated the request.

This may be the address of a proxy. It is the same as `addr` if no proxy is used. `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**

Request flags from the point of view of the original client.

This client may be behind a proxy.

struct *kr_request_qsource_flags* **comm_flags**

Request flags from the point of view of the client actually communicating with the resolver.

When PROXYv2 protocol is used, this describes the request from the proxy. When there is no proxy, this will have exactly the same value as `flags`.

size_t **size**

query packet size

int32_t **stream_id**

HTTP/2 stream ID for DoH requests.

kr_http_header_array_t **headers**

HTTP/2 headers for DoH requests.

struct *kr_request*.[anonymous] **qsource**

unsigned **rtt**

Current upstream RTT.

const struct *kr_transport* ***transport**

Current upstream transport.

struct *kr_request*.[anonymous] **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**

for logging purposes only

addr_info_f **is_tls_capable**

addr_info_f **is_tcp_connected**

addr_info_f **is_tcp_waiting**

kr_sockaddr_array_t **forwarding_targets**

When forwarding, possible targets are put here.

struct *kr_request*.[anonymous] **selection_context**

unsigned int **count_no_nsaddr**

unsigned int **count_fail_row**

alloc_wire_f **alloc_wire_cb**

CB to allocate answer wire (can be NULL).

struct *kr_extended_error* **extended_error**

EDE info; don't modify directly, use *kr_request_set_extended_error()*

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.

Param ttl
the expired TTL (i.e. it's < 0)

Return
the adjusted TTL (typically 1) or < 0.

Functions

void **kr_qflags_set**(struct *kr_qflags* *fl1, struct *kr_qflags* fl2)

Combine flags together.

This means set union for simple flags.

void **kr_qflags_clear**(struct *kr_qflags* *fl1, struct *kr_qflags* fl2)

Remove flags.

This means set-theoretic difference.

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

void **kr_rplan_deinit**(struct *kr_rplan* *rplan)

Deinitialize resolution plan, aborting any uncommitted transactions.

Parameters

- **rplan** – plan instance

bool **kr_rplan_empty**(struct *kr_rplan* *rplan)

Return true if the resolution plan is empty (i.e.

finished or initialized)

Parameters

- **rplan** – plan instance

Returns

true or false

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.

Parameters

- **rplan** – plan instance
- **parent** – query parent (or NULL)

Returns

query instance or NULL

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.

Parameters

- **rplan** – plan instance
- **parent** – query parent (or NULL)
- **name** – resolved name
- **cls** – resolved class
- **type** – resolved type

Returns

query instance or NULL

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.

Parameters

- **rplan** – plan instance
- **qry** – resolved query

Returns

0 or an error

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.

struct *kr_query* ***kr_rplan_resolved**(struct *kr_rplan* *rplan)

Return last resolved query.

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).

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.

Parameters

- **rplan** – plan instance
- **parent** – query parent (or NULL)
- **name** – resolved name
- **cls** – resolved class
- **type** – resolved type

Returns

query instance or NULL

struct **kr_qflags**

#include <rplan.h> Query flags.

Public Members

bool **NO_MINIMIZE**

Don't minimize QNAME.

bool **NO_IPV6**

Disable IPv6.

bool **NO_IPV4**

Disable IPv4.

bool **TCP**

Use TCP (or TLS) for this query.

bool **NO_ANSWER**

Do not send any answer to the client.

Request state should be set to `KR_STATE_FAIL` when this flag is set.

bool **RESOLVED**

Query is resolved.

Note that *kr_query* gets RESOLVED before following a CNAME chain; see .CNAME.

bool **AWAIT_IPV4**

Query is waiting for A address.

bool **AWAIT_IPV6**

Query is waiting for AAAA address.

bool **AWAIT_CUT**

Query is waiting for zone cut lookup.

bool **NO_EDNS**

Don't use EDNS.

bool **CACHED**

Query response is cached.

bool **NO_CACHE**

No cache for lookup; exception: finding NSs and subqueries.

bool **EXPIRING**

Query response is cached but expiring.

See is_expiring().

bool **ALLOW_LOCAL**

Allow queries to local or private address ranges.

bool **DNSSEC_WANT**

Want DNSSEC secured answer; exception: +cd, i.e.

knot_wire_get_cd(request->qsource.packet->wire)

bool **DNSSEC_BOGUS**

Query response is DNSSEC bogus.

bool **DNSSEC_INSECURE**

Query response is DNSSEC insecure.

bool **DNSSEC_CD**

Instruction to set CD bit in request.

bool **STUB**

Stub resolution, accept received answer as solved.

bool **ALWAYS_CUT**

Always recover zone cut (even if cached).

bool **DNSSEC_WEXPAND**

Query response has wildcard expansion.

bool **PERMISSIVE**

Permissive resolver mode.

bool **STRICT**

Strict resolver mode.

bool **BADCOOKIE_AGAIN**

Query again because bad cookie returned.

bool **CNAME**

Query response contains CNAME in answer section.

bool **REORDER_RR**

Reorder cached RRs.

bool **TRACE**

Also log answers on debug level.

bool **NO_0X20**

Disable query case randomization .

bool **DNSSEC_NODS**

DS non-existence is proven.

bool **DNSSEC_OPTOUT**

Closest encloser proof has optout.

bool **NONAUTH**

Non-authoritative in-bailiwick records are enough.

TODO: utilize this also outside cache.

bool **FORWARD**

Forward all queries to upstream; validate answers.

bool **DNS64_MARK**

Internal mark for dns64 module.

bool **CACHE_TRIED**

Internal to cache module.

bool **NO_NS_FOUND**

No valid NS found during last PRODUCE stage.

bool **PKT_IS_SANE**

Set by iterator in consume phase to indicate whether some basic aspects of the packet are OK, e.g.

QNAME.

bool **DNS64_DISABLE**

Don't do any DNS64 stuff (meant for view:addr).

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**

struct *kr_qflags* **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_server_selection* **server_selection**

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_query* ***initial**

The initial query (also in pending or resolved).

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).

19.6.2 Cache

Defines

TTL_MAX_MAX

Functions

int **cache_peek**(*kr_layer_t* *ctx, knot_pkt_t *pkt)

int **cache_stash**(*kr_layer_t* *ctx, knot_pkt_t *pkt)

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.

Parameters

- **cache** – cache structure to be initialized
- **api** – storage engine API
- **opts** – storage-specific options (may be NULL for default)
- **mm** – memory context.

Returns

0 or an error code

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

int **kr_cache_commit**(struct *kr_cache* *cache)

Run after a row of operations to release transaction/lock if needed.

static inline bool **kr_cache_is_open**(struct *kr_cache* *cache)

Return true if cache is open and enabled.

static inline void **kr_cache_make_checkpoint**(struct *kr_cache* *cache)

(Re)set the time pair to the current values.

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, bool ins_nsec_p)

Insert RRSet into cache, replacing any existing data.

Parameters

- **cache** – cache structure
- **rr** – inserted RRSet
- **rrsig** – RRSIG for inserted RRSet (optional)
- **rank** – rank of the data
- **timestamp** – current time (as-if; if the RR are older, their timestamp is appropriate)
- **ins_nsec_p** – update NSEC* parameters if applicable

Returns

0 or an errcode

int **kr_cache_clear**(struct *kr_cache* *cache)

Clear all items from the cache.

Parameters

- **cache** – cache structure

Returns

if nonzero is returned, there's a big problem - you probably want to abort(), perhaps except for kr_error(EAGAIN) which probably indicates transient errors.

int **kr_cache_peek_exact**(struct *kr_cache* *cache, const knot_dname_t *name, uint16_t type, struct *kr_cache_p* *peek)

int32_t **kr_cache_ttl**(const struct *kr_cache_p* *peek, const struct *kr_query* *qry, const knot_dname_t *name, uint16_t type)

int **kr_cache_materialize**(knot_rdataset_t *dst, const struct *kr_cache_p* *ref, knot_mm_t *pool)

int **kr_cache_remove**(struct *kr_cache* *cache, const knot_dname_t *name, uint16_t type)

Remove an entry from cache.

Note: only "exact hits" are considered ATM, and some other information may be removed alongside.

Parameters

- **cache** – cache structure
- **name** – dname
- **type** – rr type

Returns

number of deleted records, or negative error code

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.

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

Returns

result count or an errcode

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.

Returns

number of deleted entries or an errcode

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).

Note: timestamp is found by a syscall, and stale-serving is not considered

---

Parameters

- **is_DS** – start searching one name higher

Returns
the number of labels to remove from the name, or negative error code

int **kr_unpack_cache_key**(knot_db_val_t key, knot_dname_t *buf, uint16_t *type)

Unpack dname and type from db key.

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

Returns
length of dname or an errcode

int **kr_cache_check_health**(struct *kr_cache* *cache, int interval)

Periodic kr_cdb_api::check_health().

Parameters

- **interval** – in milliseconds. 0 for one-time check, -1 to stop the checks.

Returns
see check_health() for one-time check; otherwise normal kr_error() code.

Variables

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

kr_cdb_pt **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; enforced primarily in iterator actually.

struct timeval **checkpoint_walltime**

Wall time on the last check-point.

uint64_t **checkpoint_monotime**

Monotonic milliseconds on the last check-point.

uv_timer_t ***health_timer**

Timer used for kr_cache_check_health()

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*.[anonymous] **[anonymous]**

Header internal for cache implementation(s).

Only LMDB works for now.

Defines

KR_CACHE_KEY_MAXLEN

LATER(optim.): this is overshot, but struct key usage should be cheap ATM.

KR_CACHE_RR_COUNT_SIZE

Size of the RR count field.

VERBOSE_MSG(qry, ...)

WITH_VERBOSE(qry)

cache_op(cache, op, ...)

Shorthand for operations on cache backend.

Typedefs

typedef uint32_t **nsec_p_hash_t**

Hash of NSEC3 parameters, used as a tag to separate different chains for same zone.

typedef knot_db_val_t **entry_list_t**[*EL_LENGTH*]

Decompressed *entry_apex*.

It's an array of unparsed entry_h references. Note: arrays are passed "by reference" to functions (in C99).

Enums

enum **[anonymous]**

Values:

enumerator **ENTRY_APEX_NSECS_CNT**

enum **EL**

Indices for decompressed entry_list_t.

Values:

enumerator **EL_NS**

enumerator **EL_CNAME**

enumerator **EL_DNAME**

enumerator **EL_LENGTH**

enum **[anonymous]**

Values:

enumerator **AR_ANSWER**

Positive answer record.

It might be wildcard-expanded.

enumerator **AR_SOA**

SOA record.

enumerator **AR_NSEC**

NSEC* covering or matching the SNAME (next closer name in NSEC3 case).

enumerator **AR_WILD**

NSEC* covering or matching the source of synthesis.

enumerator **AR_CPE**

NSEC3 matching the closest provable encloser.

Functions

struct *entry_h* ***entry_h_consistent_E**(knot_db_val_t data, uint16_t type)

Check basic consistency of entry_h for 'E' entries, not looking into ->data.

(for is_packet the length of data is checked)

struct *entry_apex* ***entry_apex_consistent**(knot_db_val_t val)

static struct *entry_h* ***entry_h_consistent_NSEC**(knot_db_val_t data)

Consistency check, ATM common for NSEC and NSEC3.

static struct *entry_h* ***entry_h_consistent**(knot_db_val_t data, uint16_t type)

static inline int **nsec_p_rdlen**(const uint8_t *rdata)

static inline *nsec_p_hash_t* **nsec_p_mkHash**(const uint8_t *nsec_p)

static inline size_t **key_nwz_off**(const struct *key* *k)

static inline size_t **key_nsec3_hash_off**(const struct *key* *k)

knot_db_val_t **key_exact_type_maypkt**(struct *key* *k, uint16_t type)

Finish constructing string key for for exact search.

It's assumed that kr_dname_lf(k->buf, owner, *) had been ran.

static inline knot_db_val_t **key_exact_type**(struct *key* *k, uint16_t type)

Like key_exact_type_maypkt but with extra checks if used for RRs only.

static inline uint16_t **EL2RRTYPE**(enum *EL* i)

int **entry_h_seek**(knot_db_val_t *val, uint16_t type)

There may be multiple entries within, so rewind `val` to the one we want.

ATM there are multiple types only for the NS ktype - it also accommodates xNAMEs.

Note: `val->len` represents the bound of the whole list, not of a single entry.

Note: in case of ENOENT, `val` is still rewound to the beginning of the next entry.

Returns
error code TODO: maybe get rid of this API?

int **entry_h_splice**(knot_db_val_t *val_new_entry, uint8_t rank, const knot_db_val_t key, const uint16_t ktype, const uint16_t type, const knot_dname_t *owner, const struct *kr_query* *qry, struct *kr_cache* *cache, uint32_t timestamp)

Prepare space to insert an entry.

Some checks are performed (rank, TTL), the current entry in cache is copied with a hole ready for the new entry (old one of the same type is cut out).

Parameters

- **val_new_entry** – The only changing parameter; ->len is read, ->data written.

Returns
error code

int **entry_list_parse**(const knot_db_val_t val, *entry_list_t* list)

Parse an *entry_apex* into individual items.

Returns
error code.

static inline size_t **to_even**(size_t n)

static inline int **entry_list_serial_size**(const *entry_list_t* list)

void **entry_list_memcpy**(struct *entry_apex* *ea, *entry_list_t* list)

Fill contents of an *entry_apex*.

Note: NULL pointers are overwritten - caller may like to fill the space later.

void **stash_pkt**(const knot_pkt_t *pkt, const struct *kr_query* *qry, const struct *kr_request* *req, bool needs_pkt)

Stash the packet into cache (if suitable, etc.)

Parameters

- **needs_pkt** – we need the packet due to not stashing some RRs; see stash_rrset() for details It assumes check_dname_for_lf().

int **answer_from_pkt**(*kr_layer_t* *ctx, knot_pkt_t *pkt, uint16_t type, const struct *entry_h* *eh, const void *eh_bound, uint32_t new_ttl)

Try answering from packet cache, given an entry_h.

This assumes the TTL is OK and entry_h_consistent, but it may still return error. On success it handles all the rest, incl. qry->flags.

static inline bool **is_expiring**(uint32_t orig_ttl, uint32_t new_ttl)

Record is expiring if it has less than 1% TTL (or less than 5s)

int32_t **get_new_ttl**(const struct *entry_h* *entry, const struct *kr_query* *qry, const knot_dname_t *owner, uint16_t type, uint32_t now)

Returns signed result so you can inspect how much stale the RR is.

Note: : NSEC* uses zone name ATM; for NSEC3 the owner may not even be knowable.

Parameters

- **owner** – name for stale-serving decisions. You may pass NULL to disable stale.
- **type** – for stale-serving.

static inline int **rdataset_dematerialize_size**(const knot_rdataset_t *rds)

Compute size of serialized rdataset.

NULL is accepted as empty set.

static inline int **rdataset_dematerialized_size**(const uint8_t *data, uint16_t *rdataset_count)

Analyze the length of a dematerialized rdataset.

Note that in the data it's KR_CACHE_RR_COUNT_SIZE and then this returned size.

void **rdataset_dematerialize**(const knot_rdataset_t *rds, uint8_t *restrict data)

Serialize an rdataset.

It may be NULL as short-hand for empty.

int **entry2answer**(struct *answer* *ans, int id, const struct *entry_h* *eh, const uint8_t *eh_bound, const knot_dname_t *owner, uint16_t type, uint32_t new_ttl)

Materialize RRset + RRSIGs into ans->rrsets[id].

LATER(optim.): it's slightly wasteful that we allocate knot_rrset_t for the packet

Returns

error code. They are all bad conditions and "guarded" by kresd's assertions.

int **pkt_renew**(knot_pkt_t *pkt, const knot_dname_t *name, uint16_t type)

Prepare answer packet to be filled by RRs (without RR data in wire).

int **pkt_append**(knot_pkt_t *pkt, const struct answer_rrset *rrset, uint8_t rank)

Append RRset + its RRSIGs into the current section (*shallow* copy), with given rank.

Note: it works with empty set as well (skipped)

Note: pkt->wire is not updated in any way

Note: KNOT_CLASS_IN is assumed

Note: Whole RRsets are put into the pseudo-packet; normal parsed packets would only contain single-RR sets.

knot_db_val_t **key_NSEC1**(struct *key* *k, const knot_dname_t *name, bool add_wildcard)

Construct a string key for for NSEC (1) predecessor-search.

Note: k->zlf_len is assumed to have been correctly set

Parameters

- **add_wildcard** – Act as if the name was extended by "*."

int **nsec1_encloser**(struct *key* *k, struct *answer* *ans, const int sname_labels, int *clencl_labels, knot_db_val_t *cover_low_kwz, knot_db_val_t *cover_hi_kwz, const struct *kr_query* *qry, struct *kr_cache* *cache)

Closest encloser check for NSEC (1).

To understand the interface, see the call point.

Parameters

- **k** – space to store key + input: zname and zlf_len

Returns

0: success; >0: try other (NSEC3); <0: exit cache immediately.

int **nsec1_src_synth**(struct *key* *k, struct *answer* *ans, const knot_dname_t *clencl_name, knot_db_val_t cover_low_kwz, knot_db_val_t cover_hi_kwz, const struct *kr_query* *qry, struct *kr_cache* *cache)

Source of synthesis (SS) check for NSEC (1).

To understand the interface, see the call point.

Returns

0: continue; <0: exit cache immediately; AR_SOA: skip to adding SOA (SS was covered or matched for NODATA).

knot_db_val_t **key_NSEC3**(struct *key* *k, const knot_dname_t *nsec3_name, const *nsec_p_hash_t* nsec_p_hash)

Construct a string key for for NSEC3 predecessor-search, from an NSEC3 name.

Note: k->zlf_len is assumed to have been correctly set

int **nsec3_encloser**(struct *key* *k, struct *answer* *ans, const int sname_labels, int *clencl_labels, const struct *kr_query* *qry, struct *kr_cache* *cache)

TODO.

See nsec1_encloser(...)

int **nsec3_src_synth**(struct *key* *k, struct *answer* *ans, const knot_dname_t *clencl_name, const struct *kr_query* *qry, struct *kr_cache* *cache)

TODO.

See nsec1_src_synth(...)

static inline uint16_t **get_uint16**(const void *address)

static inline uint8_t *knot_db_val_bound(knot_db_val_t val)

Useful pattern, especially as void-pointer arithmetic isn't standard-compliant.

Variables

static const int NSEC_P_MAXLEN = sizeof(uint32_t) + 5 + 255

static const int NSEC3_HASH_LEN = 20

Hash is always SHA1; I see no plans to standardize anything else.

https://www.iana.org/assignments/dnssec-nsec3-parameters/dnssec-nsec3-parameters.xhtml#dnssec-nsec3-parameters-3

static const int NSEC3_HASH_TXT_LEN = 32

struct entry_h

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.

bool is_packet

Negative-answer packet for insecure/bogus name.

bool has_optout

Only for packets; persisted DNSSEC_OPTOUT.

uint8_t _pad

We need even alignment for data now.

uint8_t data[]

struct nsec_p

#include <*impl.h*> NSEC* parameters for the chain.

Public Members

const uint8_t ***raw**

Pointer to raw NSEC3 parameters; NULL for NSEC.

nsec_p_hash_t **hash**

Hash of `raw`, used for cache keys.

dnssec_nsec3_params_t **libknot**

Format for libknot; owns malloced memory!

struct **key**

Public Members

const knot_dname_t ***zname**

current zone name (points within qry->sname)

uint8_t **zlf_len**

length of current zone's lookup format

uint16_t **type**

Corresponding key type; e.g.

NS for CNAME. Note: NSEC type is ambiguous (exact and range key).

uint8_t **buf**[*KR_CACHE_KEY_MAXLEN*]

The key data start at buf+1, and buf[0] contains some length.

For details see key_exact* and key_NSEC* functions.

struct **entry_apex**

#include <*impl.h*> Header of 'E' entry with ktype == NS.

Inside is private to ./entry_list.c

We store xNAME at NS type to lower the number of searches in closest_NS(). CNAME is only considered for equal name, of course. We also store NSEC* parameters at NS type.

Public Members

bool **has_ns**

bool **has_cname**

bool **has_dname**

uint8_t **pad_**

1 byte + 2 bytes + x bytes would be weird; let's do 2+2+x.

int8_t **nsecs**[*ENTRY_APEX_NSECS_CNT*]

We have two slots for NSEC* parameters.

This array describes how they're filled; values: 0: none, 1: NSEC, 3: NSEC3.

Two slots are a compromise to smoothly handle normal rollovers (either changing NSEC3 parameters or between NSEC and NSEC3).

uint8_t **data**[]

struct **answer**

#include <*impl.h*> Partially constructed answer when gathering RRsets from cache.

Public Members

int **rcode**

PKT_NODATA, etc.

struct *nsec_p* **nsec_p**

Don't mix different NSEC* parameters in one answer.

knot_mm_t ***mm**

Allocator for rrsets.

struct *answer*.*answer_rrset* **rrsets**[1 + 1 + 3]

see AR_ANSWER and friends; only required records are filled

struct **answer_rrset**

Public Members

ranked_rr_array_entry_t **set**

set+rank for the main data

knot_rdataset_t **sig_rds**

RRSIG data, if any.

19.6.3 Nameservers

Provides server selection API (see *kr_server_selection*) and functions common to both implementations.

Defines

KR_NS_TIMEOUT_ROW_DEAD

KR_NS_TIMEOUT_MIN_DEAD_TIMEOUT

KR_NS_TIMEOUT_RETRY_INTERVAL

Enums

enum **kr_selection_error**

These errors are to be reported as feedback to server selection.

See *kr_server_selection::error* for more details.

Values:

enumerator **KR_SELECTION_OK**

enumerator **KR_SELECTION_QUERY_TIMEOUT**

enumerator **KR_SELECTION_TLS_HANDSHAKE_FAILED**

enumerator **KR_SELECTION_TCP_CONNECT_FAILED**

enumerator **KR_SELECTION_TCP_CONNECT_TIMEOUT**

enumerator **KR_SELECTION_REFUSED**

enumerator **KR_SELECTION_SERVFAIL**

enumerator **KR_SELECTION_FORMERR**

enumerator **KR_SELECTION_FORMERR_EDNS**

inside an answer without an OPT record

enumerator **KR_SELECTION_NOTIMPL**

with an OPT record

enumerator **KR_SELECTION_OTHER_RCODE**

enumerator **KR_SELECTION_MALFORMED**

enumerator **KR_SELECTION_MISMATCHED**

Name or type mismatch.

enumerator **KR_SELECTION_TRUNCATED**

enumerator **KR_SELECTION_DNSSEC_ERROR**

enumerator **KR_SELECTION_LAME_DELEGATION**

enumerator **KR_SELECTION_BAD_CNAME**

Too long chain, or a cycle.

enumerator **KR_SELECTION_NUMBER_OF_ERRORS**

Leave this last, as it is used as array size.

enum **kr_transport_protocol**

Values:

enumerator **KR_TRANSPORT_RESOLVE_A**

Selected name with no IPv4 address, it has to be resolved first.

enumerator **KR_TRANSPORT_RESOLVE_AAAA**

Selected name with no IPv6 address, it has to be resolved first.

enumerator **KR_TRANSPORT_UDP**

enumerator **KR_TRANSPORT_TCP**

enumerator **KR_TRANSPORT_TLS**

Functions

void **kr_server_selection_init**(struct *kr_query* *qry)

Initialize the server selection API for `qry`.

The implementation is to be chosen based on qry->flags.

int **kr_forward_add_target**(struct *kr_request* *req, const struct sockaddr *sock)

Add forwarding target to request.

This is exposed to Lua in order to add forwarding targets to request. These are then shared by all the queries in said request.

struct *kr_transport* ***select_transport**(const struct *choice* choices[], int choices_len, const struct *to_resolve* unresolved[], int unresolved_len, int timeouts, struct knot_mm *mempool, bool tcp, size_t *choice_index)

Based on passed choices, choose the next transport.

Common function to both implementations (iteration and forwarding). The `*_choose_transport` functions from `selection_*.h` preprocess the input for this one.

Parameters

- **choices** – Options to choose from, see struct above
- **unresolved** – Array of names that can be resolved (i.e. no A/AAAA record)
- **timeouts** – Number of timeouts that occurred in this query (used for exponential backoff)
- **mempool** – Memory context of current request
- **tcp** – Force TCP as transport protocol
- **choice_index** – **[out]** Optionally index of the chosen transport in the `choices` array.

Returns

Chosen transport (on mempool) or NULL when no choice is viable

void **update_rtt**(struct *kr_query* *qry, struct *address_state* *addr_state, const struct *kr_transport* *transport, unsigned rtt)

Common part of RTT feedback mechanism.

Notes RTT to global cache.

void **error**(struct *kr_query* *qry, struct *address_state* *addr_state, const struct *kr_transport* *transport, enum *kr_selection_error* sel_error)

Common part of error feedback mechanism.

struct *rtt_state* **get_rtt_state**(const uint8_t *ip, size_t len, struct *kr_cache* *cache)

Get RTT state from cache.

Returns `default_rtt_state` on unknown addresses.

Note that this opens a cache transaction which is usually closed by calling `put_rtt_state`, i.e. callee is responsible for its closing (e.g. calling kr_cache_commit).

int **put_rtt_state**(const uint8_t *ip, size_t len, struct *rtt_state* state, struct *kr_cache* *cache)

void **bytes_to_ip**(uint8_t *bytes, size_t len, uint16_t port, union *kr_sockaddr* *dst)

uint8_t ***ip_to_bytes**(const union *kr_sockaddr* *src, size_t len)

void **update_address_state**(struct *address_state* *state, union *kr_sockaddr* *address, size_t address_len, struct *kr_query* *qry)

bool **no6_is_bad**(void)

struct **kr_transport**

#include <selection.h> Output of the selection algorithm.

Public Members

knot_dname_t *ns_name

Set to "." for forwarding targets.

union *kr_sockaddr* address

size_t address_len

enum *kr_transport_protocol* protocol

unsigned timeout

Timeout in ms to be set for UDP transmission.

bool timeout_capped

Timeout was capped to a maximum value based on the other candidates when choosing this transport.

The timeout therefore can be much lower than what we expect it to be. We basically probe the server for a sudden network change but we expect it to timeout in most cases. We have to keep this in mind when noting the timeout in cache.

bool deduplicated

True iff transport was set in worker.c:subreq_finalize, that means it may be different from the one originally chosen one.

struct local_state

Public Members

int timeouts

Number of timeouts that occurred resolving this query.

bool truncated

Query was truncated, switch to TCP.

bool force_resolve

Force resolution of a new NS name (if possible) Done by selection.c:error in some cases.

bool force_udp

Used to work around auths with broken TCP.

void *private

Inner state of the implementation.

struct **kr_server_selection**

#include <selection.h> Specifies a API for selecting transports and giving feedback on the choices.

The function pointers are to be used throughout resolver when some information about the transport is obtained. E.g. RTT in `worker.c` or RCODE in `iterate.c`,...

Public Members

bool **initialized**

void (***choose_transport**)(struct *kr_query* *qry, struct *kr_transport* **transport)

Puts a pointer to next transport of `qry` to `transport` .

Allocates new *kr_transport* in request's mempool, chooses transport to be used for this query. Selection may fail, so `transport` can be set to NULL.

Param transport
to be filled with pointer to the chosen transport or NULL on failure

void (***update_rtt**)(struct *kr_query* *qry, const struct *kr_transport* *transport, unsigned rtt)

Report back the RTT of network operation for transport in ms.

void (***error**)(struct *kr_query* *qry, const struct *kr_transport* *transport, enum *kr_selection_error* error)

Report back error encountered with the chosen transport.

See `enum kr_selection`

struct *local_state* ***local_state**

struct **rtt_state**

#include <selection.h> To be held per IP address in the global LMDB cache.

Public Members

int32_t **srtt**

Smoothed RTT, i.e.

an estimate of round-trip time.

int32_t **variance**

An estimate of RTT's standard derivation (not variance).

int32_t **consecutive_timeouts**

Note: some TCP and TLS failures are also considered as timeouts.

uint64_t **dead_since**

Timestamp of pronouncing this IP bad based on KR_NS_TIMEOUT_ROW_DEAD.

struct **address_state**

#include <*selection.h*> To be held per IP address and locally "inside" query.

Public Members

unsigned int **generation**

Used to distinguish old and valid records in local_state; -1 means unusable IP.

struct *rtt_state* **rtt_state**

knot_dname_t ***ns_name**

bool **tls_capable**

int **choice_array_index**

int **error_count**

bool **broken**

int **errors**[*KR_SELECTION_NUMBER_OF_ERRORS*]

struct **choice**

#include <*selection.h*> Array of these is one of inputs for the actual selection algorithm (`select_transport`)

Public Members

union *kr_sockaddr* **address**

size_t **address_len**

struct *address_state* ***address_state**

uint16_t **port**

used to overwrite the port number; if zero, `select_transport` determines it.

struct **to_resolve**

#include <*selection.h*> Array of these is description of names to be resolved (i.e.

name without some address)

Public Members

knot_dname_t ***name**

enum *kr_transport_protocol* **type**

Either KR_TRANSPORT_RESOLVE_A or KR_TRANSPORT_RESOLVE_AAAA is valid here.

Functions

int **kr_zonecut_init**(struct *kr_zonecut* *cut, const knot_dname_t *name, knot_mm_t *pool)

Populate root zone cut with SBELT.

Parameters

- **cut** – zone cut
- **name** –
- **pool** –

Returns

0 or error code

void **kr_zonecut_deinit**(struct *kr_zonecut* *cut)

Clear the structure and free the address set.

Parameters

- **cut** – zone cut

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

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

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.

Note: addresses for names in `src` get replaced and others are left as they were.

Parameters

- **dst** – destination zone cut
- **src** – source zone cut

Returns
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…

int **kr_zonecut_copy_trust**(struct *kr_zonecut* *dst, const struct *kr_zonecut* *src)

Copy zone trust anchor and keys.

Parameters

- **dst** – destination zone cut
- **src** – source zone cut

Returns
0 or an error code

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.

Parameters

- **cut** – zone cut to be populated
- **ns** – nameserver name
- **data** – typically knot_rdata_t::data
- **len** – typically knot_rdata_t::len

Returns
0 or error code

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.

Parameters

- **cut** –
- **ns** – name server name
- **data** – typically knot_rdata_t::data
- **len** – typically knot_rdata_t::len

Returns
0 or error code

int **kr_zonecut_del_all**(struct *kr_zonecut* *cut, const knot_dname_t *ns)

Delete all addresses associated with the given name.

Parameters

- **cut** –
- **ns** – name server name

Returns
0 or error code

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.

Parameters

- **cut** –
- **ns** – name server name

Returns

pack of addresses or NULL

int **kr_zonecut_set_sbelt**(struct *kr_context* *ctx, struct *kr_zonecut* *cut)

Populate zone cut with a root zone using SBELT :rfc:1034

Parameters

- **ctx** – resolution context (to fetch root hints)
- **cut** – zone cut to be populated

Returns

0 or error code

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.

The size is limited to avoid possibility of doing too much CPU work.

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

Returns

0 or error code (ENOENT if it doesn't find anything)

bool **kr_zonecut_is_empty**(struct *kr_zonecut* *cut)

Check if any address is present in the zone cut.

Parameters

- **cut** – zone cut to check

Returns

true/false

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.

19.6.4 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 int (***kr_module_init_cb**)(struct *kr_module**)

Functions

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().

Parameters

- **module** – module structure. Will be overwritten except for ->data on success.
- **name** – module name
- **path** – module search path

Returns

0 or an error

void **kr_module_unload**(struct *kr_module* *module)

Unload module instance.

Note: currently used even for lua modules

Parameters

- **module** – module structure

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

Typedefs

typedef struct *kr_layer* **kr_layer_t**

Packet processing context.

typedef struct *kr_layer_api* **kr_layer_api_t**

Enums

enum **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:

enumerator **KR_STATE_CONSUME**

Consume data.

enumerator **KR_STATE_PRODUCE**

Produce data.

enumerator **KR_STATE_DONE**

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.

enumerator **KR_STATE_FAIL**

Error.

enumerator **KR_STATE_YIELD**

Paused, waiting for a sub-query.

Functions

static inline 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. Not called if the packet is being dropped.

void ***data**

The C module can store anything in here.

int **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**

19.6.5 Utilities

Defines

KR_STRADDR_MAXLEN

Maximum length (excluding null-terminator) of a presentation-form address returned by `kr_straddr`.

kr_require(expression)

Assert() but always, regardless of -DNDEBUG.

See also kr_assert().

kr_fails_assert(expression)

Check an assertion that's recoverable.

Return the true if it fails and needs handling.

If the check fails, optionally fork()+abort() to generate coredump and continue running in parent process. Return value must be handled to ensure safe recovery from error. Use kr_require() for unrecoverable checks. The errno variable is not mangled, e.g. you can: if (kr_fails_assert(...)) return errno;

kr_assert(expression)

Kresd assertion without a return value.

These can be turned on or off, for mandatory unrecoverable checks, use kr_require(). For recoverable checks, use kr_fails_assert().

KR_DNAME_GET_STR(dname_str, dname)

KR_RRTYPE_GET_STR(rrtype_str, rrtype)

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_request* *request, const char *msg)

Callback for request logging handler.

Param msg
[in] Log message. Pointer is not valid after handler returns.

typedef struct *kr_http_header_array_entry* **kr_http_header_array_entry_t**

typedef see_source_code **kr_http_header_array_t**

Array of HTTP headers for DoH.

typedef struct timespec **kr_timer_t**

Timer, i.e stop-watch.

Functions

void **kr_fail**(bool is_fatal, const char *expr, const char *func, const char *file, int line)

Use kr_require(), kr_assert() or kr_fails_assert() instead of directly this function.

static inline bool **kr_assert_func**(bool result, const char *expr, const char *func, const char *file, int line)

Use kr_require(), kr_assert() or kr_fails_assert() instead of directly this function.

static inline int **strcmp_p**(const void *p1, const void *p2)

A strcmp() variant directly usable for qsort() on an array of strings.

static inline void **get_workdir**(char *out, size_t len)

Get current working directory with fallback value.

char ***kr_strcatdup**(unsigned n, ...)

Concatenate N strings.

char ***kr_absolutize_path**(const char *dirname, const char *fname)

Construct absolute file path, without resolving symlinks.

Returns
malloc-ed string or NULL (+errno in that case)

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, ..)

inline uint64_t **kr_rand_bytes**(unsigned int size)

Return a few random bytes.

static inline 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

int **kr_memreserve**(void *baton, void **mem, size_t elm_size, size_t want, size_t *have)

Memory reservation routine for knot_mm_t.

int **kr_pkt_recycle**(knot_pkt_t *pkt)

int **kr_pkt_clear_payload**(knot_pkt_t *pkt)

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.

void **kr_pkt_make_auth_header**(knot_pkt_t *pkt)

Set packet header suitable for authoritative answer.

(for policy module)

static inline knot_dname_t ***kr_pkt_qname_raw**(const knot_pkt_t *pkt)

Get pointer to the in-header QNAME.

That's normally not lower-cased. However, when receiving packets from upstream we xor-apply the secret during packet-parsing, so it would get lower-cased after that point if the case was right.

const char ***kr_inaddr**(const struct sockaddr *addr)

Address bytes for given family.

int **kr_inaddr_family**(const struct sockaddr *addr)

Address family.

int **kr_inaddr_len**(const struct sockaddr *addr)

Address length for given family, i.e.

sizeof(struct in*_addr).

int **kr_sockaddr_len**(const struct sockaddr *addr)

Sockaddr length for given family, i.e.

sizeof(struct sockaddr_in*).

ssize_t **kr_sockaddr_key**(struct *kr_sockaddr_key_storage* *dst, const struct sockaddr *addr)

Creates a packed structure from the specified `addr`, safe for use as a key in containers like `trie_t`, and writes it into `dst`.

On success, returns the actual length of the key.

Returns `kr_error(EAFNOSUPPORT)` if the family of `addr` is unsupported.

struct sockaddr ***kr_sockaddr_from_key**(struct sockaddr_storage *dst, const char *key)

Creates a `struct sockaddr` from the specified `key` created using the `kr_sockaddr_key()` function.

bool **kr_sockaddr_key_same_addr**(const char *key_a, const char *key_b)

Checks whether the two keys represent the same address; does NOT compare the ports.

int **kr_sockaddr_cmp**(const struct sockaddr *left, const struct sockaddr *right)

Compare two given sockaddr.

return 0 - addresses are equal, error code otherwise.

uint16_t **kr_inaddr_port**(const struct sockaddr *addr)

Port.

void **kr_inaddr_set_port**(struct sockaddr *addr, uint16_t port)

Set port.

int **kr_inaddr_str**(const struct sockaddr *addr, char *buf, size_t *buflen)

Write string representation for given address as "<addr>#<port>".

Parameters

- **addr** – **[in]** the raw address
- **buf** – **[out]** the buffer for output string
- **buflen** – **[inout]** the available(in) and utilized(out) length, including \0

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.

char ***kr_straddr**(const struct sockaddr *addr)

int **kr_straddr_family**(const char *addr)

Return address type for string.

int **kr_family_len**(int family)

Return address length in given family (struct in*_addr).

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)

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.

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.

Returns
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

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.

void **kr_bitmask**(unsigned char *a, size_t a_len, int bits)

Masks bits.

The specified number of bits in `a` from the left (network order) will remain their original value, while the rest will be set to zero. This is useful for storing network addresses in a trie.

static inline bool **kr_sockaddr_link_local**(const struct sockaddr *addr)

Check whether `addr` points to an `AF_INET6` address and whether the address is link-local.

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.

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).

Returns
key length if successful or an error

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.

Returns
array index (>= 0) or error code (< 0)

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*))

char ***kr_pkt_text**(const knot_pkt_t *pkt)

Returns
Newly allocated string representation of packet. Caller has to free() returned string.

char ***kr_rrset_text**(const knot_rrset_t *rr)

static inline char *kr_dname_text(const knot_dname_t *name)

static inline char *kr_rrtype_text(const uint16_t rrtype)

char *kr_module_call(struct *kr_context* *ctx, const char *module, const char *prop, const char *input)

Call module property.

static inline uint16_t kr_rrset_type_maysig(const knot_rrset_t *rr)

Return the (covered) type of an nonempty RRset.

uint64_t kr_now()

The current time in monotonic milliseconds.

Note: it may be outdated in case of long callbacks; see uv_now().

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.

Returns

the number of bytes written (>0) or error code (<0)

static inline 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

static inline void kr_timer_start(*kr_timer_t* *start)

Start, i.e.

set the reference point.

static inline double kr_timer_elapsed(*kr_timer_t* *start)

Get elapsed time in floating-point seconds.

static inline uint64_t kr_timer_elapsed_us(*kr_timer_t* *start)

Get elapsed time in micro-seconds.

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)

void **kr_rrset_init**(knot_rrset_t *rrset, knot_dname_t *owner, uint16_t type, uint16_t rclass, uint32_t ttl)

bool **kr_pkt_has_wire**(const knot_pkt_t *pkt)

bool **kr_pkt_has_dnssec**(const knot_pkt_t *pkt)

uint16_t **kr_pkt_qclass**(const knot_pkt_t *pkt)

uint16_t **kr_pkt_qtype**(const knot_pkt_t *pkt)

uint32_t **kr_rrsig_sig_inception**(const knot_rdata_t *rdata)

uint32_t **kr_rrsig_sig_expiration**(const knot_rdata_t *rdata)

uint16_t **kr_rrsig_type_covered**(const knot_rdata_t *rdata)

time_t **kr_file_mtime**(const char *fname)

long long **kr_fssize**(const char *path)

Return filesystem size in bytes.

const char *__kr_dirent_name__(const struct dirent *de)

Simply return de->dname.

(useful from Lua)

Variables

static const size_t **KR_PKT_SIZE_NOWIRE** = -1

When knot_pkt is passed from cache without ->wire, this is the ->size.

bool **kr_dbg_assertion_abort**

Whether kr_assert() and kr_fails_assert() checks should abort.

int **kr_dbg_assertion_fork**

How often kr_assert() should fork the process before issuing abort (if configured).

This can be useful for debugging rare edge-cases in production. if (kr_debug_assertion_abort && kr_debug_assertion_fork), it is possible to both obtain a coredump (from forked child) and recover from the non-fatal error in the parent process.

== 0 (false): no forking

0: minimum delay between forks

(in milliseconds, each instance separately, randomized +-25%) < 0: no rate-limiting (not recommended)

const knot_dump_style_t **KR_DUMP_STYLE_DEFAULT**

Style used by the kr_*_text() functions.

struct **kr_sockaddr_key_storage**

#include <utils.h> Used for reserving enough space for the `kr_sockaddr_key` function output.

Public Members

char **bytes**[sizeof(struct sockaddr_storage)]

struct **kr_http_header_array_entry**

Public Members

char ***name**

char ***value**

union **kr_sockaddr**

#include <utils.h> Simple storage for IPx address and their ports or AF_UNSPEC.

Public Members

struct sockaddr **ip**

struct sockaddr_in **ip4**

struct sockaddr_in6 **ip6**

union **kr_in_addr**

#include <utils.h> Simple storage for IPx addresses.

Public Members

struct in_addr **ip4**

struct in6_addr **ip6**

Defines

KR_EXPORT

KR_CONST

KR_PURE

KR_NORETURN

KR_COLD

KR_PRINTF(n)

kr_ok()

kr_strerror(x)

Functions

static inline int **kr_error**(int x)

19.6.6 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.
- *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)
```

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);
```

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:

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.

Returns

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* .

Returns
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.

Returns
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.

Returns
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.

Returns
0 on success, <0 on failure

array_tail(array)

Return last element of the array.

Warning: Undefined if the array is empty.

Functions

static inline size_t **array_next_count**(size_t elm_size, size_t want, size_t have)

Choose array length when it overflows.

static inline int **array_std_reserve**(void *baton, void **mem, size_t elm_size, size_t want, size_t *have)

static inline 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);
kr_require(queue_head(q) == 2);
kr_require(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);
}
kr_require(queue_tail(q) == 5);

queue_push_head(q, 0);
++queue_tail(q);
kr_require(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 ("chunks") 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.

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 equivalent to forward-only list backed by an array.

Todo:

If some mistake happens somewhere, the access may end up in an infinite loop. (equality comparison on pointers)

Example usage:

```
pack_t pack;
pack_init(pack);

// Reserve 2 objects, 6 bytes total
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);
```

Note: Maximum object size is 2^16 bytes, see *pack_objlen_t*

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).

Returns
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* .

Returns
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.

typedef see_source_code **pack_t**

Pack is defined as an array of bytes.

Functions

static inline *pack_objlen_t* **pack_obj_len**(uint8_t *it)

Return packed object length.

static inline uint8_t ***pack_obj_val**(uint8_t *it)

Return packed object value.

static inline uint8_t ***pack_obj_next**(uint8_t *it)

Return pointer to next packed object.

static inline uint8_t ***pack_last**(*pack_t* pack)

Return pointer to the last packed object.

static inline int **pack_obj_push**(*pack_t* *pack, const uint8_t *obj, *pack_objlen_t* len)

Push object to the end of the pack.

Returns
0 on success, negative number on failure

static inline uint8_t ***pack_obj_find**(*pack_t* *pack, const uint8_t *obj, *pack_objlen_t* len)

Returns a pointer to packed object.

Returns
pointer to packed object or NULL

static inline int **pack_obj_del**(*pack_t* *pack, const uint8_t *obj, *pack_objlen_t* len)

Delete object from the pack.

Returns

0 on success, negative number on failure

static inline 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.

Returns

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;

// 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.

Parameters

- **table** – pointer to LRU
- **key_** – lookup key
- **len_** – key length

Returns
pointer to data or NULL if not found

`lru_get_new`(table, key_, len_, is_new)

Return pointer to value, inserting if needed (zeroed).

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).

Returns
pointer to data or NULL (can be even if memory could be allocated!)

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

enum **lru_apply_do**

Possible actions to do with an element.

Values:

enumerator **LRU_APPLY_DO_NOTHING**

enumerator **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.nic.cz/knot/knot-dns/tree/68352fc969/src/contrib/qp-trie only with simple adjustments, mostly include lines, KR_EXPORT and assertions.

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

trie_t ***trie_create**(knot_mm_t *mm)

Create a trie instance. Pass NULL to use malloc+free.

void **trie_free**(*trie_t* *tbl)

Free a trie instance.

void **trie_clear**(*trie_t* *tbl)

Clear a trie instance (make it empty).

size_t **trie_weight**(const *trie_t* *tbl)

Return the number of keys in the trie.

trie_val_t ***trie_get_try**(*trie_t* *tbl, const char *key, uint32_t len)

Search the trie, returning NULL on failure.

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.

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.

int **trie_get_leq**(*trie_t* *tbl, const char *key, uint32_t len, *trie_val_t* **val)

Search for less-or-equal element.

Parameters

- **tbl** – Trie.
- **key** – Searched key.
- **len** – Key length.
- **val** – Must be valid; it will be set to NULL if not found or errored.

Returns

KNOT_EOK for exact match, 1 for previous, KNOT_ENOENT for not-found, or KNOT_E*.

int **trie_apply**(*trie_t* *tbl, int (*f)(*trie_val_t**, void*), void *d)

Apply a function to every trie_val_t, in order.

Parameters

- **d** – Parameter passed as the second argument to f().

Returns

First nonzero from f() or zero (i.e. KNOT_EOK).

int **trie_apply_with_key**(*trie_t* *tbl, int (*f)(const char*, uint32_t, *trie_val_t**, void*), void *d)

Apply a function to every trie_val_t, in order.

It's like trie_apply() but additionally passes keys and their lengths.

Parameters

- **d** – Parameter passed as the second argument to f().

Returns

First nonzero from f() or zero (i.e. KNOT_EOK).

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.

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.

trie_it_t ***trie_it_begin**(*trie_t* *tbl)

Create a new iterator pointing to the first element (if any).

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).

---

bool **trie_it_finished**(*trie_it_t* *it)

Test if the iterator has gone past the last element.

void **trie_it_free**(*trie_it_t* *it)

Free any resources of the iterator. It's OK to call it on NULL.

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.

---

trie_val_t ***trie_it_val**(*trie_it_t* *it)

Return pointer to the value of the current element (writable).

CHAPTER

TWENTY

MODULES API REFERENCE

20.1 Supported languages

Currently modules written in C and Lua(JIT) are supported.

20.2 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.

[1] Mandatory symbol; defined by using *`KR_MODULE_EXPORT()`*.

20.3 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
                        return state
                end,
        finish = function (state, req, answer)
                        if state == kres.FAIL then
                                counter.failed = counter.failed + 1
                        end
                        return state
                end
```

(continues on next page)

(continued from previous page)

```
}
```

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.

20.4 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;
        int ret = pthread_join(thr_id, res);
        if (ret != 0) {
                return kr_error(errno);
        }
```

(continues on next page)

(continued from previous page)

```
        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.

20.5 Configuring modules

There is a callback `X_config()` that you can implement, see hints module.

20.6 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;
}

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
```

20.6.1 Special properties

If the module declares properties `get` or `set`, they can be used in the Lua interpreter as regular tables.

CHAPTER

TWENTYONE

WORKER API REFERENCE

Functions

int **worker_init**(struct *engine* *engine, int worker_count)

Create and initialize the worker.

Returns
error code (ENOMEM)

void **worker_deinit**(void)

Destroy the worker (free memory).

int **worker_submit**(struct *session* *session, struct io_comm_data *comm, const uint8_t *eth_from, const uint8_t *eth_to, knot_pkt_t *pkt)

Process an incoming packet (query from a client or answer from upstream).

Parameters

- **session** – session the packet came from, or NULL (not from network)
- **comm** – IO communication data (see `struct io_comm_data` docs)
- **eth_*** – MAC addresses or NULL (they're useful for XDP)
- **pkt** – the packet, or NULL (an error from the transport layer)

Returns
0 or an error code

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.

knot_pkt_t ***worker_resolve_mk_pkt_dname**(knot_dname_t *qname, uint16_t qtype, uint16_t qclass, const struct *kr_qflags* *options)

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.

struct qr_task ***worker_resolve_start**(knot_pkt_t *query, struct *kr_qflags* options)

Start query resolution with given query.

Returns
task or NULL

int **worker_resolve_exec**(struct qr_task *task, knot_pkt_t *query)

struct *kr_request* ***worker_task_request**(struct qr_task *task)

Returns
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)

struct session ***worker_find_tcp_waiting**(struct worker_ctx *worker, const struct sockaddr *addr)

struct session ***worker_find_tcp_connected**(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**(const struct *kr_request* *req)
Note: source session is NULL in case the request hasn't come over network.

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, const uv_handle_t *handle, int status)
To be called after sending a DNS message.

It mainly deals with cleanups.

Variables

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.

size_t **err_udp**

Total number of write errors for UDP transport.

size_t **err_tcp**

Total number of write errors for TCP transport.

size_t **err_tls**

Total number of write errors for TLS transport.

size_t **err_http**

Total number of write errors for HTTP(S) transport.

CHAPTER

TWENTYTWO

CUSTOM HTTP SERVICES

This chapter describes how to create custom HTTP services inside Knot Resolver. Please read HTTP module basics in chapter *Other HTTP services* before continuing.

Each network address+protocol+port combination configured using *net.listen()* is associated with *kind* of endpoint, e.g. `doh_legacy` 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
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>'}
```

22.1 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('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()
                if not tonumber(data) then
                        return 500, 'Not a good request'
                end
                value = tonumber(data)
        -- Unsupported method, return 405 Method not allowed
        else
```

(continues on next page)

(continued from previous page)

```
                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

TWENTYTHREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

V

W