Knot DNS Resolver

Release 2.4.0

CZ.NIC Labs

Jul 16, 2018

Contents

1 Knot DNS Resolver daemon **3**
- 1.1 Configuration . . . 3
- 1.2 Running supervised . . . 20
- 1.3 Enabling DNSSEC . . . 21
- 1.4 CLI interface . . . 22
- 1.5 Scaling out . . . 22
- 1.6 Using CLI tools . . . 23

2 Knot DNS Resolver modules **25**
- 2.1 Static hints . . . 26
- 2.2 Statistics collector . . . 28
- 2.3 Query policies . . . 30
- 2.4 Views and ACLs . . . 35
- 2.5 Prefetching records . . . 36
- 2.6 HTTP/2 services . . . 37
- 2.7 DNS Application Firewall . . . 42
- 2.8 Rebinding protection . . . 44
- 2.9 Graphite module . . . 45
- 2.10 Etcd module . . . 45
- 2.11 DNS64 . . . 46
- 2.12 Renumber . . . 47
- 2.13 DNS Cookies . . . 47
- 2.14 DNSSEC validation failure logging . . . 48
- 2.15 Workarounds . . . 48
- 2.16 Dnstap . . . 49
- 2.17 Signaling Trust Anchor Knowledge in DNSSEC . . . 49
- 2.18 Sentinel for Detecting Trusted Root Keys . . . 49
- 2.19 Priming module . . . 50
- 2.20 System time skew detector . . . 50
- 2.21 Detect discontinuous jumps in the system time . . . 50
- 2.22 Root on lookback (RFC 7706) . . . 50
- 2.23 Cache prefilling . . . 51
- 2.24 Serve stale . . . 51

3 Building project **53**
- 3.1 Installing from packages . . . 53
- 3.2 Platform considerations . . . 53

3.3 Requirements . . . 53
3.4 Building from sources . . . 55
3.5 Getting Docker image . . . 58

4 Knot DNS Resolver library **59**
4.1 Requirements . . . 59
4.2 For users . . . 59
4.3 For developers . . . 59
4.4 Writing layers . . . 61
4.5 APIs in Lua . . . 62
4.6 API reference . . . 65

5 Modules API reference **103**
5.1 Supported languages . . . 103
5.2 The anatomy of an extension . . . 103
5.3 Writing a module in Lua . . . 104
5.4 Writing a module in C . . . 106
5.5 Writing a module in Go . . . 106
5.6 Configuring modules . . . 108
5.7 Exposing C/Go module properties . . . 108

6 Indices and tables **111**

Knot Resolver is a minimalistic implementation of a caching validating DNS resolver. Modular architecture keeps the core tiny and efficient, and it provides a state-machine like API for extensions.

CHAPTER 1

Knot DNS Resolver daemon

The server is in the *daemon* directory, it works out of the box without any configuration.

```
$ kresd -h # Get help
$ kresd -a ::1
```

1.1 Configuration

- *Configuration example*
- *Configuration syntax*
  - *Dynamic configuration*
  - *Asynchronous events*
- *Configuration reference*
  - *Environment*
  - *Network configuration*
  - *TLS server configuration*
  - *Trust anchors and DNSSEC*
  - *Modules configuration*
  - *Cache configuration*
  - *Timers and events*
  - *Asynchronous function execution*
  - *Scripting worker*

In its simplest form the server requires just a working directory in which it can set up persistent files like cache and the process state. If you don't provide the working directory by parameter, it is going to make itself comfortable in the current working directory.

```
$ kresd /var/cache/knot-resolver
```

And you're good to go for most use cases! If you want to use modules or configure daemon behavior, read on.

There are several choices on how you can configure the daemon, a RPC interface, a CLI, and a configuration file. Fortunately all share common syntax and are transparent to each other.

1.1.1 Configuration example

```
-- interfaces
net = { '127.0.0.1', '::1' }
-- load some modules
modules = { 'policy' }
-- 10MB cache
cache.size = 10*MB
```

Tip: There are more configuration examples in *etc/* directory for personal, ISP, company internal and resolver cluster use cases.

1.1.2 Configuration syntax

The configuration is kept in the `config` file in the daemon working directory, and it's going to get loaded automatically. If there isn't one, the daemon is going to start with sane defaults, listening on *localhost*. The syntax for options is like follows: `group.option = value` or `group.action(parameters)`. You can also comment using a `--` prefix.

A simple example would be to load static hints.

```
modules = {
        'hints' -- no configuration
}
```

If the module accepts configuration, you can call the `module.config({...})` or provide options table. The syntax for table is `{ key1 = value, key2 = value }`, and it represents the unpacked JSON-encoded string, that the modules use as the *input configuration*.

```
modules = {
        hints = '/etc/hosts'
}
```

Warning: Modules specified including their configuration may not load exactly in the same order as specified.

Modules are inherently ordered by their declaration. Some modules are built-in, so it would be normally impossible to place for example *hints* before *cache*. You can enforce specific order by precedence operators **>** and **<**.

```
modules = {
   'hints  > iterate', -- Hints AFTER iterate
   'policy > hints',   -- Policy AFTER hints
   'view   < cache'    -- View BEFORE cache
}
modules.list() -- Check module call order
```

This is useful if you're writing a module with a layer, that evaluates an answer before writing it into cache for example.

Tip: The configuration and CLI syntax is Lua language, with which you may already be familiar with. If not, you can read the Learn Lua in 15 minutes for a syntax overview. Spending just a few minutes will allow you to break from static configuration, write more efficient configuration with iteration, and leverage events and hooks. Lua is heavily used for scripting in applications ranging from embedded to game engines, but in DNS world notably in PowerDNS Recursor. Knot DNS Resolver does not simply use Lua modules, but it is the heart of the daemon for everything from configuration, internal events and user interaction.

Dynamic configuration

Knowing that the the configuration is a Lua in disguise enables you to write dynamic rules. It also helps you to avoid repetitive templating that is unavoidable with static configuration.

```
if hostname() == 'hidden' then
        net.listen(net.eth0, 5353)
else
        net = { '127.0.0.1', 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 packages (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')

if cache.count() == 0 then
        -- download cache from parent
```

(continues on next page)

(continued from previous page)

```
        http.request {
                url = 'http://parent/cache.mdb',
                sink = ltn12.sink.file(io.open('cache.mdb', 'w'))
        }
        -- reopen cache with 100M limit
        cache.size = 100*MB
end
```

Asynchronous events

Lua supports a concept called closures, this is extremely useful for scripting actions upon various events, say for example - prune the cache within minute after loading, publish statistics each 5 minutes and so on. Here's an example of an anonymous function with *event.recurrent()*:

```
-- every 5 minutes
event.recurrent(5 * minute, function()
        cache.prune()
end)
```

Note that each scheduled event is identified by a number valid for the duration of the event, you may cancel it at any time. You can do this with anonymous functions, if you accept the event as a parameter, but it's not very useful as you don't have any *non-global* way to keep persistent variables.

```
-- make a closure, encapsulating counter
function pruner()
        local i = 0
        -- pruning function
        return function(e)
                cache.prune()
                -- cancel event on 5th attempt
                i = i + 1
                if i == 5 then
                        event.cancel(e)
                fi
        end
end

-- make recurrent event that will cancel after 5 times
event.recurrent(5 * minute, pruner())
```

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 HTTP servers or monitoring probes that cooperate well with the daemon internal operations. See *event.socket()*

File watchers are possible with *worker.coroutine()* and cqueues, see the cqueues documentation for more information.

```
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
```

(continues on next page)

(continued from previous page)

```
      print(name, notify[flag])
    end
  end
end)
```

1.1.3 Configuration reference

This is a reference for variables and functions available to both configuration file and CLI.

- *Environment*
- *Network configuration*
- *TLS server configuration*
- *Trust anchors and DNSSEC*
- *Modules configuration*
- *Cache configuration*
- *Timers and events*
- *Asynchronous function execution*
- *Scripting worker*

Environment

env (table)
Return environment variable.

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

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 affects ephemeral certificates for kresd serving DNS over TLS.

moduledir ([dir])

Returns Modules directory.

If called with a parameter, it will change kresd's directory for looking up the dynamic modules. If called without a parameter, it will return kresd's modules directory.

verbose (true | *false*)

Returns Toggle verbose logging.

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

Returns 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

reorder_RR ([true | *false*])

Parameters

- **value** (*boolean*) – New value for the option *(optional)*

Returns The (new) value of the option

If set, resolver will vary the order of resource records within RR-sets every time when answered from cache. It is disabled by default.

user (name, [group])

Parameters

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

Returns boolean

Drop privileges and run 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 = { '127.0.0.1', '::1' }
-- unprivileged
cache.size = 100*MB
trust_anchors.file = 'root.key'
```

Example output:

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

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

Parameters

- **name** (*string*) – Query name (e.g. 'com.')

[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.

- **type** (*number*) – Query type (e.g. `kres.type.NS`)
- **class** (*number*) – Query class *(optional)* (e.g. `kres.class.IN`)
- **options** (*number*) – Resolution options (see query flags)
- **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.

Returns boolean

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 (answer, req)
   -- Check answer RCODE
   local pkt = kres.pkt_t(answer)
   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)
```

package_version()

Returns Current package version.

This returns current package version (the version of the binary) as a string.

```
> package_version()
2.1.1
```

Network configuration

For when listening on `localhost` just doesn't cut it.

Tip: Use declarative interface for network.

```
net = { '127.0.0.1', net.eth0, net.eth1.addr[1] }
net.ipv4 = false
```

net.ipv6 = true|false

Return boolean (default: true)

Enable/disable using IPv6 for recursion.

net.ipv4 = true|false

Return boolean (default: true)

Enable/disable using IPv4 for recursion.

net.listen(addresses, [port = *53*, flags = {tls = (port == *853)}])

Returns boolean

Listen on addresses; port and flags are optional. The addresses can be specified as a string or device, or a list of addresses (recursively). The command can be given multiple times, but note that it silently skips any addresses that have already been bound.

Examples:

```
net.listen('::1')
net.listen(net.lo, 5353)
net.listen({net.eth0, '127.0.0.1'}, 53853, {tls = true})
```

net.close(address, [port = *53*])

Returns boolean

Close opened address/port pair, noop if not listening.

net.list()

Returns Table of bound interfaces.

Example output:

```
[127.0.0.1] => {
    [port] => 53
    [tcp] => true
    [udp] => true
}
```

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
```

(continues on next page)

(continued from previous page)

```
}
[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.bufsize ([udp_bufsize])
Get/set maximum EDNS payload available. Default is 4096. You cannot set less than 512 (512 is DNS packet size without EDNS, 1220 is minimum size for DNSSEC) or more than 65535 octets.

Example output:

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

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
```

net.outgoing_v4 ([string *address]*)
Get/set the IPv4 address used to perform queries. There is also `net.outgoing_v6` for IPv6. The default is `nil`, which lets the OS choose any address.

TLS server configuration

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

Example output:

```
> net.tls("/etc/knot-resolver/server-cert.pem", "/etc/knot-resolver/server-key.pem
↪")
> net.tls()
("/etc/knot-resolver/server-cert.pem", "/etc/knot-resolver/server-key.pem")
> net.listen("::", 853)
> net.listen("::", 443, {tls = true})
```

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

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

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

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

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

> **Warning: Setting the secret is probably too risky with TLS <= 1.2**. At this moment no GnuTLS stable release even supports TLS 1.3. Therefore setting the secrets should be considered experimental for now and might not be available on your system.

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

Trust anchors and DNSSEC

trust_anchors.config (keyfile, readonly)
Alias for *add_file*. It is also equivalent to CLI parameter `-k <keyfile>` and `trust_anchors.file = keyfile`.

trust_anchors.add_file (keyfile, readonly)

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.

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

Example output:

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

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

trust_anchors.hold_down_time = 30 * day

Return int (default: 30 * day)

Modify RFC5011 hold-down timer to given value. 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. 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 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.

Tip: Use the *trust_anchors.negative = {}* alias for easier configuration.

Example output:

```
> trust_anchors.negative = { '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.

Example output:

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

Modules configuration

The daemon provides an interface for dynamic loading of *daemon modules*.

Tip: Use declarative interface for module loading.

```
modules = {
        hints = {file = '/etc/hosts'}
}
```

Equals to:

```
modules.load('hints')
hints.config({file = '/etc/hosts'})
```

---

modules.list()

Returns List of loaded modules.

modules.load(name)

Parameters

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

Returns boolean

Load a module by name.

modules.unload(name)

Parameters

- **name** (*string*) – Module name

Returns boolean

Unload a module by name.

Cache configuration

The default cache in Knot DNS Resolver is persistent with LMDB backend, this means that the daemon doesn't lose the cached data on restart or crash to avoid cold-starts. The cache may be reused between cache daemons or manipulated from other processes, making for example synchronised load-balanced recursors possible.

cache.size (number)
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 (number)
Get the maximum size in bytes.

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

cache.storage (string)
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 (string)
Get the storage backend configuration.

```
print(cache.storage)
```

cache.backends()

Returns map of 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.stats()

return table of cache counters

The cache collects counters on various operations (hits, misses, transactions, ...). This function call returns a table of cache counters that can be used for calculating statistics.

cache.open(max_size[, config_uri])

Parameters

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

Returns boolean

Open cache with 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 cache supports runtime-changeable backends, see *cache.backends()* for mor information and default. Refer to specific documentation of specific backends for configuration string syntax.

- `lmdb://`

As of now it only allows you to change the cache directory, e.g. `lmdb:///tmp/cachedir`.

cache.count()

Returns Number of entries in the cache or nil on error.

cache.close()

Returns boolean

Close the cache.

Note: This may or may not clear the cache, depending on the used backend. See *cache.clear()*.

cache.stats()

Return table of statistics, note that this tracks all operations over cache, not just which queries were answered from cache or not.

Example:

```
print('Insertions:', cache.stats().insert)
```

cache.max_ttl([ttl])

Parameters

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

Returns current maximum TTL

Get or set maximum cache TTL.

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

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

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

cache.min_ttl([ttl])

Parameters

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

Returns current maximum TTL

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

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

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

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

cache.ns_tout([timeout])

Parameters

- **timeout** – number of 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.prune ([max_count])

Parameters

- **max_count** (*number*) – maximum number of items to be pruned at once (default: 65536)

Returns `{ pruned: int }`

Prune expired/invalid records.

cache.get ([domain])

Returns list of matching records in cache

Fetches matching records from cache. The **domain** can either be:

- a domain name (e.g. `"domain.cz"`)
- a wildcard (e.g. `"*.domain.cz"`)

The domain name fetches all records matching this name, while the wildcard matches all records at or below that name.

You can also use a special namespace `"P"` to purge NODATA/NXDOMAIN matching this name (e.g. `"domain.cz P"`).

Note: This is equivalent to `cache['domain']` getter.

Examples:

```
-- Query cache for 'domain.cz'
cache['domain.cz']
-- Query cache for all records at/below 'insecure.net'
cache['*.insecure.net']
```

cache.clear ([domain])

Returns `bool`

Purge cache records. If the domain isn't provided, whole cache is purged. See *cache.get()* documentation for subtree matching policy.

Examples:

```
-- Clear records at/below 'bad.cz'
cache.clear('*.bad.cz')
-- Clear packet cache
cache.clear('*. P')
-- Clear whole cache
cache.clear()
```

Timers and events

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

event.after (time, function)

Returns event id

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

Example:

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

event.recurrent (interval, function)

Returns event id

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

Example:

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

event.reschedule (event_id, timeout)

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

Example:

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

event.cancel (event_id)

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

Example:

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

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

event.socket (fd, cb)

Parameters

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

Returns event id

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

Example:

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

Asynchronous function execution

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

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

Example:

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

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

When daemon is running in forked mode, each process acts independently. This is good because it reduces software complexity and allows for runtime scaling, but not ideal because of additional operational burden. For example, when you want to add a new policy, you'd need to add it to either put it in the configuration, or execute command on each process independently. The daemon simplifies this by promoting process group leader which is able to execute commands synchronously over forks.

Example:

```
worker.sleep(1)
```

map (expr)
Run expression synchronously over all forks, results are returned as a table ordered as forks. Expression can be any valid expression in Lua.

Example:

```
-- Current instance only
hostname()
localhost
-- Mapped to forks
map 'hostname()'
[1] => localhost
[2] => localhost
-- Get worker ID from each fork
map 'worker.id'
[1] => 0
```

(continues on next page)

(continued from previous page)

```
[2] => 1
-- Get cache stats from each fork
map 'cache.stats()'
[1] => {
    [hit] => 0
    [delete] => 0
    [miss] => 0
    [insert] => 0
}
[2] => {
    [hit] => 0
    [delete] => 0
    [miss] => 0
    [insert] => 0
}
```

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.count
Return current total worker count (e.g. *1* for single-process)

worker.id
Return current worker ID (starting from *0* up to *worker.count - 1*)

worker.pid
Current worker process PID (number).

worker.stats()
Return table of statistics.

- `udp` - number of outbound queries over UDP
- `tcp` - number of outbound queries over TCP
- `ipv6` - number of outbound queries over IPv6
- `ipv4` - number of outbound queries over IPv4
- `timeout` - number of timeouted outbound queries
- `concurrent` - number of concurrent queries at the moment
- `queries` - number of inbound queries
- `dropped` - number of dropped inbound queries

Example:

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

1.2 Running supervised

Knot Resolver can run under a supervisor to allow for graceful restarts, watchdog process and socket activation. This way the supervisor binds to sockets and lends them to the resolver daemon. If the resolver terminates or is killed, the

sockets remain open and no queries are dropped.

The watchdog process must notify kresd about active file descriptors, and kresd will automatically determine the socket type and bound address, thus it will appear as any other address. You should have a look at real process managers.

The daemon also supports systemd socket activation, it is automatically detected and requires no configuration on users's side.

See `kresd.systemd(7)` for details.

1.3 Enabling DNSSEC

The resolver supports DNSSEC including **RFC 5011** automated DNSSEC TA updates and **RFC 7646** negative trust anchors. To enable it, you need to provide trusted root keys. Bootstrapping of the keys is automated, and kresd fetches root trust anchors set over a secure channel from IANA. From there, it can perform **RFC 5011** automatic updates for you.

Note: Automatic bootstrap requires luasocket and luasec installed.

```
$ kresd -k root-new.keys # File for root keys
[ ta ] keyfile 'root-new.keys': doesn't exist, bootstrapping
[ ta ] Root trust anchors bootstrapped over https with pinned certificate.
       You SHOULD verify them manually against original source:
       https://www.iana.org/dnssec/files
[ ta ] Current root trust anchors are:
. 0 IN DS 19036 8 2 49AAC11D7B6F6446702E54A1607371607A1A41855200FD2CE1CDDE32F24E8FB5
. 0 IN DS 20326 8 2 E06D44B80B8F1D39A95C0B0D7C65D08458E880409BBC683457104237C7F8EC8D
[ ta ] next refresh for . in 24 hours
```

Alternatively, you can set it in configuration file with `trust_anchors.file = 'root.keys'`. If the file doesn't exist, it will be automatically populated with root keys validated using root anchors retrieved over HTTPS.

This is equivalent to using unbound-anchor:

```
$ unbound-anchor -a "root.keys" || echo "warning: check the key at this point"
$ echo "auto-trust-anchor-file: \"root.keys\"" >> unbound.conf
$ unbound -c unbound.conf
```

Warning: Bootstrapping of the root trust anchors is automatic, you are however **encouraged to check** the key over **secure channel**, as specified in DNSSEC Trust Anchor Publication for the Root Zone. This is a critical step where the whole infrastructure may be compromised, you will be warned in the server log.

Configuration is described in *Trust anchors and DNSSEC*.

1.3.1 Manually providing root anchors

The root anchors bootstrap may fail for various reasons, in this case you need to provide IANA or alternative root anchors. The format of the keyfile is the same as for Unbound or BIND and contains DS/DNSKEY records.

1. Check the current TA published on IANA website
2. Fetch current keys (DNSKEY), verify digests

3. Deploy them

```
$ kdig DNSKEY . @k.root-servers.net +noall +answer | grep "DNSKEY[[:space:]]257" >␣
↪root.keys
$ ldns-key2ds -n root.keys # Only print to stdout
... verify that digest matches TA published by IANA ...
$ kresd -k root.keys
```

You've just enabled DNSSEC!

Note: Bootstrapping and automatic update need write access to keyfile direcory. If you want to manage root anchors manually you should use `trust_anchors.add_file('root.keys', true)`.

1.4 CLI interface

The daemon features a CLI interface, type `help()` to see the list of available commands.

```
$ kresd /var/cache/knot-resolver
[system] started in interactive mode, type 'help()'
> cache.count()
53
```

1.4.1 Verbose output

If the verbose logging is compiled in, i.e. not turned off by `-DNOVERBOSELOG`, you can turn on verbose tracing of server operation with the `-v` option. You can also toggle it on runtime with `verbose(true|false)` command.

```
$ kresd -v
```

To run the daemon by hand, such as under `nohup`, use `-f 1` to start a single fork. For example:

```
$ nohup ./daemon/kresd -a 127.0.0.1 -f 1 -v &
```

1.5 Scaling out

The server can clone itself into multiple processes upon startup, this enables you to scale it on multiple cores. Multiple processes can serve different addresses, but still share the same working directory and cache. You can add, start and stop processes during runtime based on the load.

```
$ kresd -f 4 rundir > kresd.log &
$ kresd -f 2 rundir > kresd_2.log & # Extra instances
$ pstree $$ -g
bash(3533)─┬─kresd(19212)─┬─kresd(19212)
           │              ├─kresd(19212)
           │              └─kresd(19212)
           ├─kresd(19399)───kresd(19399)
           └─pstree(19411)
$ kill 19399 # Kill group 2, former will continue to run
bash(3533)─┬─kresd(19212)─┬─kresd(19212)
```

(continues on next page)

(continued from previous page)

```
           │                ├─kresd(19212)
           │                └─kresd(19212)
           └─pstree(19460)
```

Note: On recent Linux supporting `SO_REUSEPORT` (since 3.9, backported to RHEL 2.6.32) it is also able to bind to the same endpoint and distribute the load between the forked processes. If your OS doesn't support it, use only one daemon process.

Notice the absence of an interactive CLI. You can attach to the the consoles for each process, they are in `rundir/tty/PID`.

```
$ nc -U rundir/tty/3008 # or socat - UNIX-CONNECT:rundir/tty/3008
> cache.count()
53
```

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

This is also a way to enumerate and test running instances, the list of files in `tty` corresponds to the list of running processes, and you can test the process for liveliness by connecting to the UNIX socket.

1.6 Using CLI tools

- `kresd-host.lua` - a drop-in replacement for *host(1)* utility

Queries the DNS for information. The hostname is looked up for IP4, IP6 and mail.

Example:

```
$ kresd-host.lua -f root.key -v nic.cz
nic.cz. has address 217.31.205.50 (secure)
nic.cz. has IPv6 address 2001:1488:0:3::2 (secure)
nic.cz. mail is handled by 10 mail.nic.cz. (secure)
nic.cz. mail is handled by 20 mx.nic.cz. (secure)
nic.cz. mail is handled by 30 bh.nic.cz. (secure)
```

- `kresd-query.lua` - run the daemon in zero-configuration mode, perform a query and execute given callback.

This is useful for executing one-shot queries and hooking into the processing of the result, for example to check if a domain is managed by a certain registrar or if it's signed.

Example:

```
$ kresd-query.lua www.sub.nic.cz 'assert(kres.dname2str(req:resolved().zone_cut.name)␣
↪== "nic.cz.")' && echo "yes"
yes
$ kresd-query.lua -C 'trust_anchors.config("root.keys")' nic.cz
↪'assert(req:resolved().flags.DNSSEC_WANT)'
$ echo $?
0
```

CHAPTER 2

Knot DNS Resolver modules

- *Static hints*
- *Statistics collector*
- *Query policies*
- *Views and ACLs*
- *Prefetching records*
- *HTTP/2 services*
- *DNS Application Firewall*
- *Rebinding protection*
- *Graphite module*
- *Etcd module*
- *DNS64*
- *Renumber*
- *DNS Cookies*
- *DNSSEC validation failure logging*
- *Workarounds*
- *Dnstap*
- *Signaling Trust Anchor Knowledge in DNSSEC*
- *Sentinel for Detecting Trusted Root Keys*
- *Priming module*
- *System time skew detector*

- *Detect discontinuous jumps in the system time*
- *Root on lookback (RFC 7706)*
- *Cache prefilling*
- *Serve stale*

2.1 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.

2.1.1 Examples

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

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

2.1.2 Properties

`hints.config`([path])

Parameters

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

Returns `{ result:  bool }`

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

`hints.add_hosts`([path])

Parameters

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

Add hints from a host-like file.

`hints.get`(*hostname*)

Parameters

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

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

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

hints.set(pair)

Parameters

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

Returns `{ result: bool }`

Add a hostname - address pair hint.

Note: If multiple addresses have been added for a name, 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()

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

Tip: If no parameters are passed, returns current root hints set.

---

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.

Example:

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

(continues on next page)

(continued from previous page)

```
  [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, NODATA will be synthesised for matching hint name, but mismatching type (e.g. AAAA query when only A hint exists).

2.2 Statistics collector

This modules gathers various counters from the query resolution and server internals, and offers them as a key-value storage. Any module may update the metrics or simply hook in new ones.

```
-- 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

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

-- 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)
```

(continues on next page)

(continued from previous page)

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

2.2.1 Properties

stats.get(key)

Parameters

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

Returns `number`

Return nominal value of given metric.

stats.set(key, val)

Parameters

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

Set nominal value of given metric.

stats.list([prefix])

Parameters

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

Outputs collected metrics as a JSON dictionary.

stats.upstreams()

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

stats.frequent()

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

stats.clear_frequent()

Clear the list of most frequent iterative queries.

2.2.2 Built-in statistics

- `answer.total` - total number of answered queries

- `answer.cached` - number of queries answered from cache
- `answer.noerror` - number of **NOERROR** answers
- `answer.nodata` - number of **NOERROR**, but empty answers
- `answer.nxdomain` - number of **NXDOMAIN** answers
- `answer.servfail` - number of **SERVFAIL** answers
- `answer.1ms` - number of answers completed in 1ms
- `answer.10ms` - number of answers completed in 10ms
- `answer.50ms` - number of answers completed in 50ms
- `answer.100ms` - number of answers completed in 100ms
- `answer.250ms` - number of answers completed in 250ms
- `answer.500ms` - number of answers completed in 500ms
- `answer.1000ms` - number of answers completed in 1000ms
- `answer.1500ms` - number of answers completed in 1500ms
- `answer.slow` - number of answers that took more than 1500ms
- `query.edns` - number of queries with EDNS
- `query.dnssec` - number of queries with DNSSEC DO=1

2.3 Query policies

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

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

By default, if no rule applies to a query, built-in rules for special-use and locally-served domain names are applied. These built-in rules can be overriden using action `PASS`, see *Policy examples* below.

2.3.1 Filters

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

- `all(action)` - always applies the action
- `pattern(action, pattern)` - applies the action if QNAME matches a regular expression
- `suffix(action, table)` - applies the action if QNAME suffix matches one of suffixes in the table (useful for "is domain in zone" rules), uses Aho-Corasick string matching algorithm from CloudFlare (BSD 3-clause)
- *`policy.suffix_common`*
- `rpz(default_action, path)` - implements a subset of RPZ in zonefile format. See below for details: *`policy.rpz`*.

- custom filter function

2.3.2 Actions

An *action* is function which modifies DNS query. There are several actions available in the `policy.` table:

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

Most actions stop the policy matching on the query, but "chain actions" allow to keep trying to match other rules, until a non-chain action is triggered.

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

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

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

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

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

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

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

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 exactly match hostname in server's certificate, i.e. in most cases it must not contain trailing dot (*res.example.com*).
- System CA certificate store will be used if no *ca_file* option is specified.
- Optional *ca_file* option can specify path to CA certificate (or certificate bundle) in PEM format.

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'}
})
```

2.3.4 Policy examples

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

(continues on next page)

(continued from previous page)

```
policy.add(policy.suffix(policy.DENY, {todname('badboy.cz.')}))

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

-- Disallow ANY queries
policy.add(function (req, query)
        if query.stype == kres.type.ANY then
                return policy.DROP
        end
end)
-- Enforce local RPZ
policy.add(policy.rpz(policy.DENY, 'blacklist.rpz'))
-- Forward all queries below 'company.se' to given resolver
policy.add(policy.suffix(policy.FORWARD('192.168.1.1'), {todname('company.se')}))
-- Forward all queries matching pattern
policy.add(policy.pattern(policy.FORWARD('2001:DB8::1'), '\4bad[0-9]\2cz'))
-- Forward all queries (to public resolvers https://www.nic.cz/odvr)
policy.add(policy.all(policy.FORWARD({'2001:678:1::206', '193.29.206.206'})))
-- Print all responses with matching suffix
policy.add(policy.suffix(policy.QTRACE, {todname('rhybar.cz.')}))
-- Print all responses
policy.add(policy.all(policy.QTRACE))
-- Mirror all queries and retrieve information
local rule = policy.add(policy.all(policy.MIRROR('127.0.0.2')))
-- Print information about the rule
print(string.format('id: %d, matched queries: %d', rule.id, rule.count)
-- Reroute all addresses found in answer from 192.0.2.0/24 to 127.0.0.x
-- this policy is enforced on answers, therefore 'postrule'
local rule = policy.add(policy.REROUTE({'192.0.2.0/24', '127.0.0.0'}), true)
-- Delete rule that we just created
policy.del(rule.id)
```

2.3.5 Additional properties

Most properties (actions, filters) are described above.

policy.add(rule, postrule)

Parameters

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

Returns rule description

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

policy.del (id)

Parameters

- **id** – identifier of a given rule

Returns boolean

Remove a rule from policy list.

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

Parameters

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

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

policy.rpz (action, path)

Parameters

- **action** – the default action for match in the zone; typically you want `policy.DENY`
- **path** – path to zone file | database

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

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

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

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

Param names table of domain names in textual format

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

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

```
-- 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' }
```

This module is enabled by default because it implements mandatory **RFC 6761** logic. For debugging purposes you can add `modules.unload('policy')` to your config to unload the module.

2.4 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, sort of like ISC BIND views.

There are two identification mechanisms:

- `addr` - identifies the client based on his subnet
- `tsig` - identifies the client based on a TSIG key

You can combine this information with *policy* rules.

```
view:addr('10.0.0.1', policy.suffix(policy.TC, {'\7example\3com'}))
```

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

2.4.1 Example configuration

```
-- Load modules
modules = { 'policy', 'view' }
-- Whitelist queries identified by TSIG key
view:tsig('\5mykey', function (req, qry) return policy.PASS end)
-- Block local clients (ACL like)
view:addr('127.0.0.1', function (req, qry) return policy.DENY end))
-- Drop queries with suffix match for remote client
view:addr('10.0.0.0/8', policy.suffix(policy.DROP, {'\3xxx'}))
-- RPZ for subset of clients
view:addr('192.168.1.0/24', policy.rpz(policy.PASS, 'whitelist.rpz'))
-- Forward all queries from given subnet to proxy
view:addr('10.0.0.0/8', policy.all(policy.FORWARD('2001:DB8::1')))
-- Drop everything that hasn't matched
view:addr('0.0.0.0/0', function (req, qry) return policy.DROP end)
```

2.4.2 Properties

view:addr (subnet, rule)

Parameters

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

Apply rule to clients in given subnet.

view:tsig (key, rule)

Parameters

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

Apply rule to clients with given TSIG key.

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

2.5 Prefetching records

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

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

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

2.5.1 Example configuration

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

Defaults are 15 minutes window, 6 hours period.

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

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

2.5.2 Exported metrics

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

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

2.5.3 Properties

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

2.6 HTTP/2 services

This is a module that does the heavy lifting to provide an HTTP/2 enabled server that supports TLS by default and provides endpoint for other modules in order to enable them 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.

The server allows other modules to either use default endpoint that provides built-in webpage, restful APIs and websocket streams, or create new endpoints.

2.6.1 Example configuration

By default, the web interface starts HTTPS/2 on port 8053 using an ephemeral certificate that is valid for 90 days and is automatically renewed. It is of course self-signed, so you should use your own judgement before exposing it to the outside world. Why not use something like Let's Encrypt for starters?

```
-- Load HTTP module with defaults
modules = {
        http = {
                host = 'localhost', -- Default: 'localhost'
                port = 8053,        -- Default: 8053
                geoip = 'GeoLite2-City.mmdb' -- Optional, see
                -- e.g. https://dev.maxmind.com/geoip/geoip2/geolite2/
                -- and install mmdblua library
                endpoints = {},
        }
}
```

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

```
$ curl -k https://localhost:8053
$ curl -k https://localhost:8053/stats
```

It is possible to disable HTTPS altogether by passing `cert = false` option. While it's not recommended, it 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.

```
http = {
        cert = false,
}
```

If you want to provide your own certificate and key, you're welcome to do so:

```
http = {
        cert = 'mycert.crt',
        key  = 'mykey.key',
}
```

The format of both certificate and key is expected to be PEM, e.g. equivallent 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
```

2.6.2 Built-in services

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

Endpoint	Service	Description
`/stats`	Statistics/metrics	Exported metrics in JSON.
`/metrics`	Prometheus metrics	Exported metrics for Prometheus
`/feed`	Most frequent queries	List of most frequent queries in JSON.
`/trace/:name/:type`	Tracking	Trace resolution of the query and return the verbose logs.

2.6.3 Enabling Prometheus metrics endpoint

The module exposes `/metrics` endpoint that serves internal metrics in Prometheus text format. You can use it out of the box:

```
$ curl -k https://localhost:8053/metrics | tail
# TYPE latency histogram
latency_bucket{le=10} 2.000000
latency_bucket{le=50} 2.000000
latency_bucket{le=100} 2.000000
latency_bucket{le=250} 2.000000
latency_bucket{le=500} 2.000000
latency_bucket{le=1000} 2.000000
latency_bucket{le=1500} 2.000000
latency_bucket{le=+Inf} 2.000000
latency_count 2.000000
latency_sum 11.000000
```

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

```
http = {
        host = 'localhost',
}

-- Set Prometheus namespace
http.prometheus.namespace = 'resolver_'
```

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

```
http = {
        host = 'localhost',
}

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

2.6.4 Tracing requests

With the `/trace` endpoint you can trace various aspects of the request execution. The basic mode allows you to resolve a query and trace verbose logs (and messages received):

```
$ curl http://localhost:8053/trace/e.root-servers.net
[ 8138] [iter] 'e.root-servers.net.' type 'A' created outbound query, parent id 0
[ 8138] [ rc ] => rank: 020, lowest 020, e.root-servers.net. A
[ 8138] [ rc ] => satisfied from cache
[ 8138] [iter] <= answer received:
;; ->>HEADER<<- opcode: QUERY; status: NOERROR; id: 8138
;; Flags: qr aa  QUERY: 1; ANSWER: 0; AUTHORITY: 0; ADDITIONAL: 0

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

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

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

2.6.5 How to expose services over HTTP

The module provides a table `endpoints` of already existing endpoints, it is free for reading and writing. It contains tables describing a triplet - `{mime, on_serve, on_websocket}`. In order to register a new service, simply add it to the table:

```
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}
-- Load module
```

(continues on next page)

(continued from previous page)

```
modules = {
        http = {
                endpoints = { ['/health'] = on_health }
        }
}
```

Then you can query the API endpoint, or tail the WebSocket using curl.

```
$ curl -k http://localhost:8053/health
{"state":"up","uptime":0}
$ curl -k -i -N -H "Connection: Upgrade" -H "Upgrade: websocket" -H "Host:␣
↪localhost:8053/health"  -H "Sec-Websocket-Key: nope" -H "Sec-Websocket-Version: 13"␣
↪https://localhost:8053/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. 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>'}
```

2.6.6 How to expose RESTful services

A RESTful service is likely to respond differently to different type of methods and requests, there are three things that you can do in a service handler to send back results. First is to just send whatever you want to send back, it has to respect MIME type that the service declared in the endpoint definition. The response code would then be `200 OK`, any non-string responses will be packed to JSON. Alternatively, you can respond with a number corresponding to the HTTP response code or send headers and body yourself.

```
-- Our upvalue
local value = 42

-- Expose the service
local service = {'application/json',
function (h, stream)
        -- Get request method and deal with it properly
        local m = h:get(':method')
        local path = h:get(':path')
        log('[service] method %s path %s', m, path)
        -- Return table, response code will be '200 OK'
        if m == 'GET' then
                return {key = path, value = value}
        -- Save body, perform check and either respond with 505 or 200 OK
        elseif m == 'POST' then
                local data = stream:get_body_as_string()
                if not tonumber(data) then
                        return 500, 'Not a good request'
```

(continues on next page)

(continued from previous page)

```
                end
                value = tonumber(data)
        -- Unsupported method, return 405 Method not allowed
        else
                return 405, 'Cannot do that'
        end
end}
-- Load the module
modules = {
        http = {
                endpoints = { ['/service'] = service }
        }
}
```

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
```

2.6.7 How to expose more interfaces

Services exposed in the previous part share the same external interface. This means that it's either accessible to the outside world or internally, but not one or another. This is not always desired, i.e. you might want to offer DNS/HTTPS to everyone, but allow application firewall configuration only on localhost. `http` module allows you to create additional interfaces with custom endpoints for this purpose.

```
http.add_interface {
        endpoints = {
                ['/conf'] = {
                        'application/json', function (h, stream)
                                return 'configuration API\n'
                        end
                },
        },
        -- Same options as the config() method
        host = 'localhost',
        port = '8054',
}
```

This way you can have different internal-facing and external-facing services at the same time.

2.6.8 Dependencies

- lua-http (>= 0.1) available in LuaRocks

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

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

  Any other system can install from LuaRocks directly:

  ```
  $ luarocks install http
  ```

- mmdblua available in LuaRocks

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

2.7 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.

2.7.1 Example configuration

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

2.7.2 Web interface

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

2.7.3 RESTful interface

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

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

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

```
# Get current rule set
$ curl -s -X GET http://localhost:8053/daf | jq .
{}

# Create new rule
$ curl -s -X POST -d "src = 127.0.0.1 pass" http://localhost:8053/daf | jq .
{
  "count": 0,
  "active": true,
  "info": "src = 127.0.0.1 pass",
  "id": 1
}

# Disable rule
$ curl -s -X PATCH http://localhost:8053/daf/1/active/false | jq .
true

# Retrieve a rule information
$ curl -s -X GET http://localhost:8053/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:8053/daf/1 | jq .
true
```

2.8 Rebinding protection

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

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

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

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

Warning: Some like to "misuse" such addresses, e.g. *127.*.*.** in blacklists served over DNS, and this module will block such uses.

2.9 Graphite module

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

2.9.1 Example configuration

Only the `host` parameter is mandatory.

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

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

The module supports sending data to multiple servers at once.

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

2.9.2 Dependencies

- luasocket available in LuaRocks

```
$ luarocks install luasocket
```

2.10 Etcd module

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

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

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

Configures all listening nodes to following configuration:

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

2.10.1 Example configuration

```
modules = {
        etcd = {
                prefix = '/knot-resolver',
                peer = 'http://127.0.0.1:7001'
        }
}
```

Warning: Work in progress!

2.10.2 Dependencies

- lua-etcd available in LuaRocks

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

2.11 DNS64

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

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

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

2.11.1 Example configuration

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

2.12 Renumber

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.

2.12.1 Example configuration

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

2.13 DNS Cookies

The module performs most of the RFC 7873 DNS cookies functionality. Its main purpose is to check the cookies of inbound queries and responses. It is also used to alter the behaviour of the cookie functionality.

2.13.1 Example Configuration

```
-- Load the module before the 'iterate' layer.
modules = {
        'cookies < iterate'
}

-- Configure the client part of the resolver. Set 8 bytes of the client
-- secret and choose the hashing algorithm to be used.
-- Use a string composed of hexadecimal digits to set the secret.
cookies.config { client_secret = '0123456789ABCDEF',
                 client_cookie_alg = 'FNV-64' }

-- Configure the server part of the resolver.
cookies.config { server_secret = 'FEDCBA9876543210',
                  server_cookie_alg = 'FNV-64' }

-- Enable client cookie functionality. (Add cookies into outbound
-- queries.)
cookies.config { client_enabled = true }

-- Enable server cookie functionality. (Handle cookies in inbound
-- requests.)
cookies.config { server_enabled = true }
```

Tip: If you want to change several parameters regarding the client or server configuration then do it within a single `cookies.config()` invocation.

Warning: The module must be loaded before any other module that has direct influence on query processing and response generation. The module must be able to intercept an incoming query before the processing of the actual query starts. It must also be able to check the cookies of inbound responses and eventually discard them before they are handled by other functional units.

2.13.2 Properties

cookies.config(configuration)

Parameters

- **configuration** (*table*) – part of cookie configuration to be changed, may be called without parameter

Returns JSON dictionary containing current configuration

The function may be called without any parameter. In such case it only returns current configuration. The returned JSON also contains available algorithm choices.

2.13.3 Dependencies

- Nettle required for HMAC-SHA256

2.14 DNSSEC validation failure logging

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

Add following line to your configuration file to enable it:

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

Example of error message logged by this module:

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

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

2.15 Workarounds

A simple module that alters resolver behavior on specific broken sub-domains. Currently it mainly disables case randomization on them.

2.15.1 Running

```
modules = { 'workarounds < iterate' }
```

2.16 Dnstap

Dnstap module currently supports logging dns responses to a unix socket in dnstap format using fstrm framing library. The unix socket and the socket reader should be present before starting kresd.

2.16.1 Configuration

Tunables:

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

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

2.17 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.

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

2.18 Sentinel for Detecting Trusted Root Keys

The module implementing A Root Key Trust Anchor Sentinel for DNSSEC according to draft-ietf-dnsop-kskroll-sentinel-12.

This feature allows users of validating resolver to detect which root keys are configured in their chain of trust. The data from such signaling are necessary to monitor the progress of the DNSSEC root key rollover.

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.

2.19 Priming module

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

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

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

2.20 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.

2.21 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.

2.22 Root on lookback (RFC 7706)

Knot Resolver developers decided that pure implementation of **RFC 7706** 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 *prefill* and *serve_stale* modules with Aggressive Use of DNSSEC-Validated Cache (**RFC 8198**) behavior which is enabled automatically together with DNSSEC validation.

2.23 Cache prefilling

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

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

Example configuration is:

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

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

Root zone to import must be signed using DNSSEC and the resolver must have valid DNSSEC configuration. (For further details please see *Enabling DNSSEC*.)

Parameter	Description
ca_file	path to CA certificate bundle used to authenticate the HTTPS connection
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.

2.23.1 Dependencies

Depends on the luasec library.

2.24 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`.

2.24.1 Running

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

CHAPTER 3

Building project

3.1 Installing from packages

The resolver is packaged for Debian, Fedora+EPEL, Ubuntu, Docker, NixOS/NixPkgs, FreeBSD, HomeBrew, and Turris Omnia. Some of these are maintained directly by the knot-resolver team.

Refer to project page for information about installing from packages. If packages are not available for your OS, see following sections to see how you can build it from sources (or package it), or use official Docker images.

3.2 Platform considerations

Knot-resolver is written for UNIX-like systems, mainly in C99. Portable I/O is provided by libuv. 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.

Windows systems might theoretically work without large changes, but it's most likely broken and currently not planned to be supported.

3.3 Requirements

The following is a list of software required to build Knot DNS Resolver from sources.

Requirement	Required by	Notes
GNU Make 3.80+	*all*	*(build only)*
C and C++ compiler	*all*	*(build only)*[1]
pkg-config	*all*	*(build only)*[2]
hexdump or xxd	`daemon`	*(build only)*
libknot 2.6.7+	*all*	Knot DNS libraries - requires autotools, GnuTLS, ...
LuaJIT 2.0+	`daemon`	Embedded scripting language.
libuv 1.7+	`daemon`	Multiplatform I/O and services (libuv 1.0 with limitations[3]).
lmdb	`daemon`	If missing, a static version is embedded.

There are also *optional* packages that enable specific functionality in Knot DNS Resolver, they are useful mainly for developers to build documentation and tests.

Optional	Needed for	Notes
lua-http	`modules/http`	HTTP/2 client/server for Lua.
luasocket	`trust anchors`, `modules/stats`	Sockets for Lua.
luasec	`trust anchors`	TLS for Lua.
cmocka	`unit tests`	Unit testing framework.
Doxygen	`documentation`	Generating API documentation.
Sphinx and sphinx_rtd_theme	`documentation`	Building this HTML/PDF documentation.
breathe	`documentation`	Exposing Doxygen API doc to Sphinx.
libsystemd	`daemon`	Systemd socket activation support.
libprotobuf 3.0+	`modules/dnstap`	Protocol Buffers support for dnstap.
libprotobuf-c 1.0+	`modules/dnstap`	C bindings for Protobuf.
libfstrm 0.2+	`modules/dnstap`	Frame Streams data transport protocol.
luacheck	`lint-lua`	Syntax and static analysis checker for Lua.
clang-tidy	`lint-c`	Syntax and static analysis checker for C.
luacov	`check-config`	Code coverage analysis for Lua modules.

3.3.1 Packaged dependencies

Most of the dependencies can be resolved from packages, here's an overview for several platforms.

- **Debian** (since *sid*) - current stable doesn't have libknot and libuv, which must be installed from sources.

```
sudo apt-get install pkg-config libknot-dev libuv1-dev libcmocka-dev libluajit-5.1-dev
```

- **Ubuntu** - unknown.
- **Fedora**

```
# minimal build
sudo dnf install @buildsys-build knot-devel libuv-devel luajit-devel
# unit tests
sudo dnf install libcmocka-devel
# integration tests
sudo dnf install cmake git python-dns python-jinja2
# optional features
sudo dnf install golang hiredis-devel libmemcached-devel lua-sec-compat lua-socket-
↪compat systemd-devel
# docs
sudo dnf install doxygen python-breathe python-sphinx
```

- **RHEL/CentOS** - unknown.
- **openSUSE** - there is an experimental package.

[1] Requires C99, `__attribute__((cleanup))` and `-MMD -MP` for dependency file generation. GCC, Clang and ICC are supported.

[2] You can use variables `<dependency>_CFLAGS` and `<dependency>_LIBS` to configure dependencies manually (i.e. `libknot_CFLAGS` and `libknot_LIBS`).

[3] libuv 1.7 brings SO_REUSEPORT support that is needed for multiple forks. libuv < 1.7 can be still used, but only in single-process mode. Use *different method* for load balancing.

- **FreeBSD** - when installing from ports, all dependencies will install automatically, corresponding to the selected options.
- **NetBSD** - unknown.
- **OpenBSD** - unknown.
- **Mac OS X** - the dependencies can be found through Homebrew.

```
brew install pkg-config libuv luajit cmocka
```

3.4 Building from sources

Initialize git submodules first.

```
$ git submodule update --init --recursive
```

The Knot DNS Resolver depends on the the Knot DNS library, recent version of libuv, and LuaJIT.

```
$ make info # See what's missing
```

When you have all the dependencies ready, you can build and install.

```
$ make PREFIX="/usr/local"
$ make install PREFIX="/usr/local"
```

Note: Always build with `PREFIX` if you want to install, as it is hardcoded in the executable for module search path. Production code should be compiled with `-DNDEBUG`. If you build the binary with `-DNOVERBOSELOG`, it won't be possible to turn on verbose logging; we advise packagers against using that flag.

Note: If you build with `PREFIX`, you may need to also set the `LDFLAGS` for the libraries:

```
make LDFLAGS="-Wl,-rpath=/usr/local/lib" PREFIX="/usr/local"
```

Alternatively you can build only specific parts of the project, i.e. `library`.

```
$ make lib
$ make lib-install
```

Note: Documentation is not built by default, run `make doc` to build it.

3.4.1 Building with security compiler flags

Knot DNS Resolver enables certain security compile-time flags that do not affect performance. You can add more flags to the build by appending them to *CFLAGS* variable, e.g. `make CFLAGS="-fstack-protector"`.

Method	Status	Notes
-fstack-protector	*disabled*	(must be specifically enabled in CFLAGS)
-D_FORTIFY_SOURCE=2	**enabled**	
-pie	**enabled**	enables ASLR for kresd (disable with `make HARDENING=no`)
RELRO	**enabled**	full[4]

You can also disable linker hardening when it's unsupported with `make HARDENING=no`.

3.4.2 Building for packages

The build system supports DESTDIR

```
$ make install DESTDIR=/tmp/stage
```

Tip: There is a template for service file and AppArmor profile to help you kickstart the package.

3.4.3 Default paths

The default installation follows FHS with several custom paths for configuration and modules. All paths are prefixed with `PREFIX` variable by default if not specified otherwise.

Component	Variable	Default	Notes
library	`LIBDIR`	`$(PREFIX)/lib`	pkg-config is auto-generated[5]
daemon	`SBINDIR`	`$(PREFIX)/sbin`	
configuration	`ETCDIR`	`$(PREFIX)/etc/knot-resolver`	Configuration file, templates.
modules	`MODULEDIR`	`$(LIBDIR)/kdns_modules`	Runtime directory for loading dynamic modules[6].
trust anchor file	`KEYFILE_DEFAULT`	*(none)*	Path to read-only trust anchor file, which is used as fallback when no other file is specified.[7]
work directory		the current directory	Run directory for daemon. (Only relevant during run time, not e.g. during installation.)

Note: Each module is self-contained and may install additional bundled files within `$(MODULEDIR)/$(modulename)`. These files should be read-only, non-executable.

[4] See checksec.sh

[5] The `libkres.pc` is installed in `$(LIBDIR)/pkgconfig`.

[6] The default moduledir can be changed with *-m* option to *kresd* daemon or by calling *moduledir()* function from lua.

[7] If no other trust anchor is specified by user, the compiled-in path `KEYFILE_DEFAULT` must contain a valid trust anchor. This is typically used by distributions which provide DNSSEC root trust anchors as part of distribution package. Users can disable the built-in trust anchor by adding `trust_anchors.keyfile_default = nil` to their configuration.

3.4.4 Static or dynamic?

By default the resolver library is built as a dynamic library with versioned ABI. You can revert to static build with `BUILDMODE` variable.

```
$ make BUILDMODE=dynamic # Default, create dynamic library
$ make BUILDMODE=static  # Create static library
```

When the library is linked statically, it usually produces a smaller binary. However linking it to various C modules might violate ODR and increase the size.

3.4.5 Resolving dependencies

The build system relies on pkg-config to find dependencies. You can override it to force custom versions of the software by environment variables.

```
$ make libknot_CFLAGS="-I/opt/include" libknot_LIBS="-L/opt/lib -lknot -ldnssec"
```

Optional dependencies may be disabled as well using `HAS_x=yes|no` variable.

```
$ make HAS_go=no HAS_cmocka=no
```

Warning: If the dependencies lie outside of library search path, you need to add them somehow. Try `LD_LIBRARY_PATH` on Linux/BSD, and `DYLD_FALLBACK_LIBRARY_PATH` on OS X. Otherwise you need to add the locations to linker search path.

Several dependencies may not be in the packages yet, the script pulls and installs all dependencies in a chroot. You can avoid rebuilding dependencies by specifying *BUILD_IGNORE* variable, see the Dockerfile for example. Usually you only really need to rebuild libknot.

```
$ export FAKEROOT="${HOME}/.local"
$ export PKG_CONFIG_PATH="${FAKEROOT}/lib/pkgconfig"
$ export BUILD_IGNORE="..." # Ignore installed dependencies
$ ./scripts/bootstrap-depends.sh ${FAKEROOT}
```

3.4.6 Building extras

The project can be built with code coverage tracking using the `COVERAGE=1` variable.

The *make coverage* target gathers both gcov code coverage for C files, and luacov code coverage for Lua files and merges it for analysis. It requires lcov to be installed.

```
$ make coverage
```

3.4.7 Running unit and integration tests

The linter requires luacheck and clang-tidy and is executed by `make lint`. The unit tests require cmocka and are executed by `make check`. Tests for the dnstap module need go and are executed by `make ckeck-dnstap`.

The integration tests use Deckard, the DNS test harness.

```
$  make check-integration
```

Note that the daemon and modules must be installed first before running integration tests, the reason is that the daemon is otherwise unable to find and load modules.

Read the documentation for more information about requirements, how to run it and extend it.

3.5 Getting Docker image

Docker images require only either Linux or a Linux VM (see boot2docker on OS X).

```
$ docker run cznic/knot-resolver
```

See the Docker images page for more information and options. You can hack on the container by changing the container entrypoint to shell like:

```
$ docker run -it --entrypoint=/bin/bash cznic/knot-resolver
```

Tip: You can build the Docker image yourself with `docker build -t knot-resolver scripts`.

CHAPTER 4

Knot DNS Resolver library

4.1 Requirements

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

4.2 For users

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

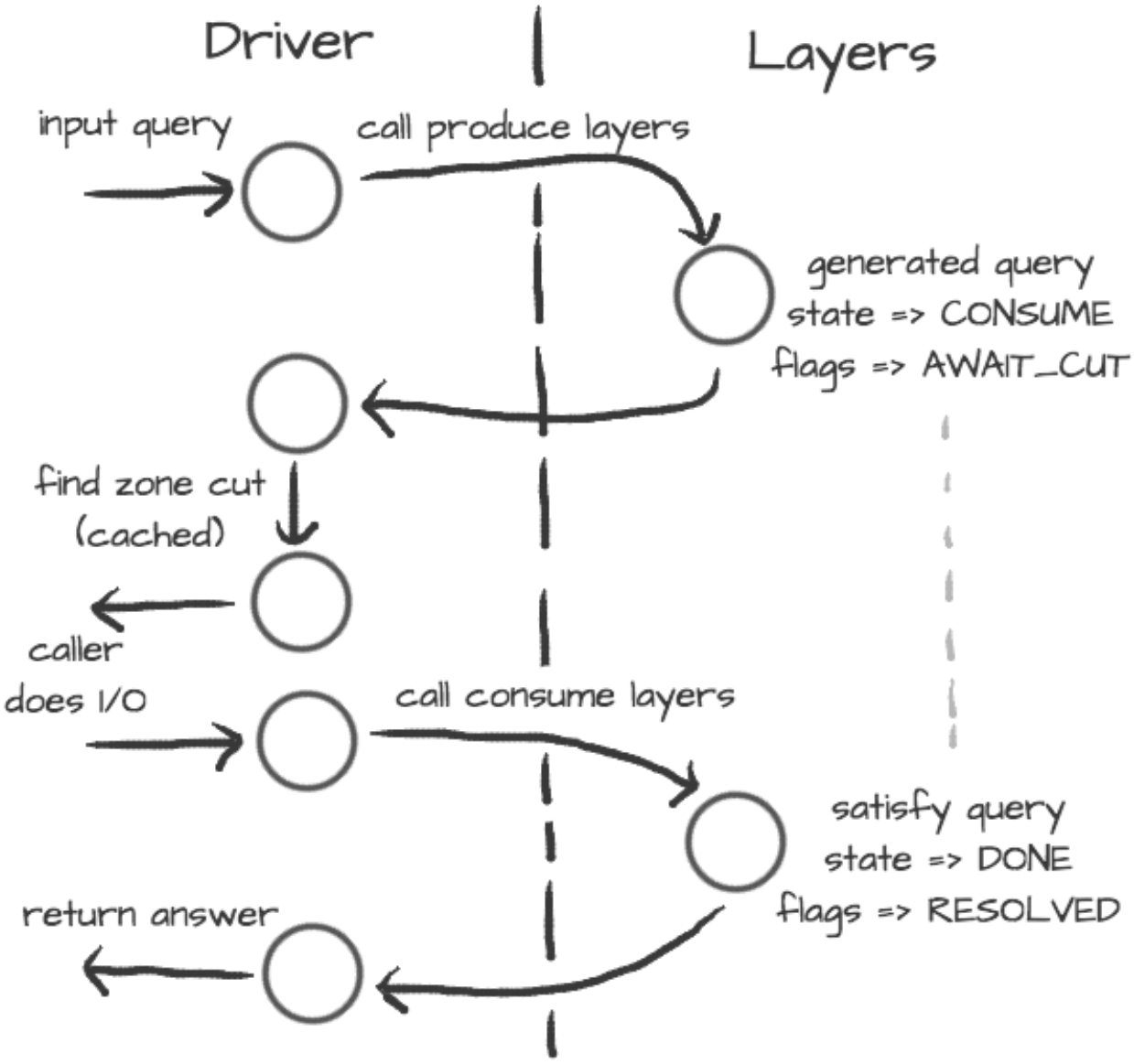
Tip: If you're migrating from `getaddrinfo()`, see *"synchronous"* API, but the library offers iterative API as well to plug it into your event loop for example.

4.3 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)
        answer = kres.pkt_t(answer)
        if answer:qtype() == kres.type.NS then
```

(continues on next page)

(continued from previous page)

```
                req = kres.request_t(req)
                local qry = req:push(answer:qname(), kres.type.SOA, kres.class.IN)
                qry.flags.AWAIT_CUT = true
        end
        return state
end
```

This **doesn't** block currently processed query, and the newly created sub-request will start as soon as driver finishes processing current. In some cases you might need to issue sub-request and process it **before** continuing with the current, i.e. validator may need a DNSKEY before it can validate signatures. In this case, layers can yield and resume afterwards.

```
consume = function (state, req, answer)
        answer = kres.pkt_t(answer)
        if state == kres.YIELD then
                print('continuing yielded layer')
                return kres.DONE
        else
                if answer:qtype() == kres.type.NS then
                        req = kres.request_t(req)
                        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.

4.4 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;
}
```

4.5 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.

4.5.1 Elementary types and constants

- States are directly in `kres` table, e.g. `kres.YIELD`, `kres.CONSUME`, `kres.PRODUCE`, `kres.DONE`, `kres.FAIL`.
- DNS classes are in `kres.class` table, e.g. `kres.class.IN` for Internet class.
- DNS types are in `kres.type` table, e.g. `kres.type.AAAA` for AAAA type.
- DNS rcodes types are in `kres.rcode` table, e.g. `kres.rcode.NOERROR`.
- Packet sections (QUESTION, ANSWER, AUTHORITY, ADDITIONAL) are in the `kres.section` table.

4.5.2 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)
```

4.5.3 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))
```

4.5.4 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)
        pkt = kres.pkt_t(answer)
        print('rcode:', pkt:rcode())
        print('query:', kres.dname2str(pkt:qname()), pkt:qclass(), pkt:qtype())
        if pkt:rcode() ~= kres.rcode.NOERROR then
                print('error response')
```

(continues on next page)

(continued from previous page)

```
        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)
```

4.5.5 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)
        req = kres.request_t(req)
        print(req.options)
        print(req.state)

        -- Print information about current query
        local current = req:current()
        print(kres.dname2str(current.owner))
        print(current.stype, current.sclass, current.id, current.flags)
end
```

In layers that either begin or finalize, you can walk the list of resolved queries.

```
local last = req:resolved()
print(last.stype)
```

As described in the layers, you can not only retrieve information about current query, but also push new ones or pop old ones.

```
-- Push new query
local qry = req:push(pkt:qname(), kres.type.SOA, kres.class.IN)
qry.flags.AWAIT_CUT = true

-- Pop the query, this will erase it from resolution plan
req:pop(qry)
```

4.6 API reference

- *Name resolution*
- *Cache*
- *Nameservers*
- *Modules*
- *Utilities*
- *Generics library*

4.6.1 Name resolution

The API provides an API providing a "consumer-producer"-like interface to enable user to plug it into existing event loop or I/O code.

Example usage of the iterative API:

```
// Create request and its memory pool
struct kr_request req = {
    .pool = {
        .ctx = mp_new (4096),
        .alloc = (mm_alloc_t) mp_alloc
    }
};

// Setup and provide input query
int state = kr_resolve_begin(&req, ctx, final_answer);
state = kr_resolve_consume(&req, query);

// Generate answer
while (state == KR_STATE_PRODUCE) {

    // Additional query generate, do the I/O and pass back answer
    state = kr_resolve_produce(&req, &addr, &type, query);
    while (state == KR_STATE_CONSUME) {
        int ret = sendrecv(addr, proto, query, resp);

        // If I/O fails, make "resp" empty
        state = kr_resolve_consume(&request, addr, resp);
        knot_pkt_clear(resp);
    }
    knot_pkt_clear(query);
```

(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.

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:

KR_RANK_INITIAL = 0
Did not attempt to validate.

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

KR_RANK_OMIT
Do not attempt to validate.

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

KR_RANK_TRY
Attempt to validate, but failures are non-fatal.

KR_RANK_INDET = 4
Unable to determine whether it should be secure.

KR_RANK_BOGUS
Ought to be secure but isn't.

KR_RANK_MISMATCH

KR_RANK_MISSING
Unable to obtain a good signature.

KR_RANK_INSECURE = 8
Proven to be insecure, i.e.

we have a chain of trust from TAs that cryptographically denies the possibility of existence of a positive chain of trust from the TAs to the record.

KR_RANK_AUTH = 16
Authoritative data flag; the chain of authority was "verified".

Even if not set, only in-bailiwick stuff is acceptable, i.e. almost authoritative (example: mandatory glue and its NS RR).

KR_RANK_SECURE = 32
Verified whole chain of trust from the closest TA.

Functions

bool **kr_rank_check**(uint8_t *rank*)
Check that a rank value is valid.

Meant for assertions.

static bool **kr_rank_test**(uint8_t *rank*, uint8_t *kr_flag*)
Test the presence of any flag/state in a rank, i.e.

including KR_RANK_AUTH.

static void **kr_rank_set**(uint8_t * *rank*, uint8_t *kr_flag*)
Set the rank state.

The _AUTH flag is kept as it was.

KR_EXPORT int **kr_resolve_begin**(struct *kr_request* * *request*, struct *kr_context* * *ctx*, knot_pkt_t * *answer*)
Begin name resolution.

Note Expects a request to have an initialized mempool, the "answer" packet will be kept during the resolution and will contain the final answer at the end.

Return CONSUME (expecting query)

Parameters

- `request`: request state with initialized mempool
- `ctx`: resolution context
- `answer`: allocated packet for final answer

KR_EXPORT int **kr_resolve_consume**(struct *kr_request* * *request*, const struct sockaddr * *src*, knot_pkt_t * *packet*)
Consume input packet (may be either first query or answer to query originated from *kr_resolve_produce()*)

Note If the I/O fails, provide an empty or NULL packet, this will make iterator recognize nameserver failure.

Return any state

Parameters

- `request`: request state (awaiting input)
- `src`: [in] packet source address
- `packet`: [in] input packet

KR_EXPORT int **kr_resolve_produce**(struct *kr_request* * *request*, struct sockaddr ** *dst*, int * *type*, knot_pkt_t * *packet*)
Produce either next additional query or finish.

If the CONSUME is returned then dst, type and packet will be filled with appropriate values and caller is responsible to send them and receive answer. If it returns any other state, then content of the variables is undefined.

Return any state

Parameters

- `request`: request state (in PRODUCE state)
- `dst`: [out] possible address of the next nameserver
- `type`: [out] possible used socket type (SOCK_STREAM, SOCK_DGRAM)
- `packet`: [out] packet to be filled with additional query

KR_EXPORT int **kr_resolve_checkout** (struct *kr_request* * *request*, struct sockaddr * *src*, struct sockaddr * *dst*, int *type*, knot_pkt_t * *packet*)
Finalises the outbound query packet with the knowledge of the IP addresses.

Note The function must be called before actual sending of the request packet.

Return kr_ok() or error code

Parameters

- `request`: request state (in PRODUCE state)
- `src`: address from which the query is going to be sent
- `dst`: address of the name server
- `type`: used socket type (SOCK_STREAM, SOCK_DGRAM)
- `packet`: [in,out] query packet to be finalised

KR_EXPORT int **kr_resolve_finish** (struct *kr_request* * *request*, int *state*)
Finish resolution and commit results if the state is DONE.

Note The structures will be deinitialized, but the assigned memory pool is not going to be destroyed, as it's owned by caller.

Return DONE

Parameters

- `request`: request state
- `state`: either DONE or FAIL state

KR_EXPORT KR_PURE struct *kr_rplan** **kr_resolve_plan** (struct *kr_request* * *request*)
Return resolution plan.

Return pointer to rplan

Parameters

- `request`: request state

KR_EXPORT KR_PURE knot_mm_t* **kr_resolve_pool** (struct *kr_request* * *request*)
Return memory pool associated with request.

Return mempool

Parameters

- `request`: request state

struct kr_context

#include <resolve.h> Name resolution context.

Resolution context provides basic services like cache, configuration and options.

Note This structure is persistent between name resolutions and may be shared between threads.

Public Members

struct *kr_qflags* **options**

knot_rrset_t* **opt_rr**

map_t **trust_anchors**

map_t **negative_anchors**

struct *kr_zonecut* **root_hints**

struct *kr_cache* **cache**

kr_nsrep_rtt_lru_t* **cache_rtt**

unsigned **cache_rtt_tout_retry_interval**

kr_nsrep_lru_t* **cache_rep**

module_array_t* **modules**

struct kr_cookie_ctx **cookie_ctx**

kr_cookie_lru_t* **cache_cookie**

int32_t **tls_padding**
See net.tls_padding in ../daemon/README.rst -1 is "true" (default policy), 0 is "false" (no padding)

knot_mm_t* **pool**

struct kr_request

#include <resolve.h> Name resolution request.

Keeps information about current query processing between calls to processing APIs, i.e. current resolved query, resolution plan, ... Use this instead of the simple interface if you want to implement multiplexing or custom I/O.

Note All data for this request must be allocated from the given pool.

Public Members

struct *kr_context** **ctx**

knot_pkt_t* **answer**

struct *kr_query** **current_query**
Current evaluated query.

const knot_rrset_t* **key**

const struct sockaddr* **addr**
Address that originated the request.

Current upstream address.

NULL for internal origin.

const struct sockaddr* **dst_addr**
Address that accepted the request.

NULL for internal origin.

const knot_pkt_t* **packet**

const knot_rrset_t* **opt**

bool **tcp**
true if the request is on tcp; only meaningful if (dst_addr)

size_t **size**
query packet size

struct *kr_request*::@6 **qsource**

unsigned **rtt**
Current upstream RTT.

struct *kr_request*::@7 **upstream**
Upstream information, valid only in consume() phase.

struct *kr_qflags* **options**

int **state**

ranked_rr_array_t **answ_selected**

ranked_rr_array_t **auth_selected**

ranked_rr_array_t **add_selected**

rr_array_t **additional**

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**

int **has_tls**

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**

Typedefs

typedef int32_t **(* kr_stale_cb)** (int32_t *ttl*, const knot_dname_t **owner*, uint16_t *type*, const struct *kr_query* **qry*)
Callback for serve-stale decisions.

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

Parameters

- `ttl`: the expired TTL (i.e. it's < 0)

Functions

KR_EXPORT void **kr_qflags_set** (struct *kr_qflags* * *fl1*, struct *kr_qflags* *fl2*)
Combine flags together.

This means set union for simple flags.

KR_EXPORT void **kr_qflags_clear** (struct *kr_qflags* * *fl1*, struct *kr_qflags* *fl2*)
Remove flags.

This means set-theoretic difference.

KR_EXPORT int **kr_rplan_init** (struct *kr_rplan* * *rplan*, struct *kr_request* * *request*, knot_mm_t * *pool*)
Initialize resolution plan (empty).

Parameters

- `rplan`: plan instance
- `request`: resolution request
- `pool`: ephemeral memory pool for whole resolution

KR_EXPORT void **kr_rplan_deinit** (struct *kr_rplan* * *rplan*)
Deinitialize resolution plan, aborting any uncommited transactions.

Parameters

- `rplan`: plan instance

KR_EXPORT KR_PURE bool **kr_rplan_empty** (struct *kr_rplan* * *rplan*)
Return true if the resolution plan is empty (i.e.

finished or initialized)

Return true or false

Parameters

- `rplan`: plan instance

KR_EXPORT struct *kr_query** **kr_rplan_push_empty** (struct *kr_rplan* * *rplan*, struct *kr_query* * *parent*)
Push empty query to the top of the resolution plan.

Note This query serves as a cookie query only.

Return query instance or NULL

Parameters

- `rplan`: plan instance
- `parent`: query parent (or NULL)

KR_EXPORT struct *kr_query** **kr_rplan_push** (struct *kr_rplan* * *rplan*, struct *kr_query* * *parent*, const knot_dname_t * *name*, uint16_t *cls*, uint16_t *type*)
Push a query to the top of the resolution plan.

Note This means that this query takes precedence before all pending queries.

Return query instance or NULL

Parameters

- `rplan`: plan instance
- `parent`: query parent (or NULL)
- `name`: resolved name
- `cls`: resolved class
- `type`: resolved type

KR_EXPORT int **kr_rplan_pop** (struct *kr_rplan* * *rplan*, struct *kr_query* * *qry*)
Pop existing query from the resolution plan.

Note Popped queries are not discarded, but moved to the resolved list.

Return 0 or an error

Parameters

- `rplan`: plan instance
- `qry`: resolved query

KR_EXPORT KR_PURE bool **kr_rplan_satisfies** (struct *kr_query* * *closure*, const knot_dname_t * *name*, uint16_t *cls*, uint16_t *type*)
Return true if resolution chain satisfies given query.

KR_EXPORT KR_PURE struct *kr_query** **kr_rplan_resolved** (struct *kr_rplan* * *rplan*)
Return last resolved query.

KR_EXPORT KR_PURE struct *kr_query** **kr_rplan_last** (struct *kr_rplan* * *rplan*)
Return last query (either currently being solved or last resolved).

This is necessary to retrieve the last query in case of resolution failures (e.g. time limit reached).

KR_EXPORT KR_PURE struct *kr_query** **kr_rplan_find_resolved** (struct *kr_rplan* * *rplan*, struct *kr_query* * *parent*, const knot_dname_t * *name*, uint16_t *cls*, uint16_t *type*)
Check if a given query already resolved.

Return query instance or NULL

Parameters

- `rplan`: plan instance
- `parent`: query parent (or NULL)
- `name`: resolved name
- `cls`: resolved class
- `type`: resolved type

struct kr_qflags
#include <*rplan.h*> Query flags.

Public Members

bool **NO_MINIMIZE**
Don't minimize QNAME.

bool **NO_THROTTLE**
No query/slow NS throttling.

bool **NO_IPV6**
Disable IPv6.

bool **NO_IPV4**
Disable IPv4.

bool **TCP**
Use TCP for this query.

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 **SAFEMODE**
Don't use fancy stuff (EDNS, 0x20, . . .)

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.

bool **ALLOW_LOCAL**
Allow queries to local or private address ranges.

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

knot_wire_set_cd(request->answer->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 if verbose.

bool **NO_0X20**
Disable query case randomization .

bool **DNSSEC_NODS**
DS non-existance 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.

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**

struct *kr_qflags* flags **forward_flags**

uint32_t **secret**

uint16_t **fails**

uint16_t **reorder**
Seed to reorder (cached) RRs in answer or zero.

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**

uint32_t **uid**
Query iteration number, unique within the *kr_rplan*.

struct *kr_query** **cname_parent**
Pointer to the query that originated this one because of following a CNAME (or NULL).

struct *kr_request** **request**
Parent resolution request.

kr_stale_cb **stale_cb**
See the type.

struct *kr_nsrep* **ns**

struct kr_rplan
#include <*rplan.h*> Query resolution plan structure.

The structure most importantly holds the original query, answer and the list of pending queries required to resolve the original query. It also keeps a notion of current zone cut.

Public Members

kr_qarray_t **pending**
List of pending queries.

kr_qarray_t **resolved**
List of resolved queries.

struct *kr_request** **request**
Parent resolution request.

knot_mm_t* **pool**
Temporary memory pool.

uint32_t **next_uid**
Next value for *kr_query::uid* (incremental).

4.6.2 Cache

Functions

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

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

KR_EXPORT int **kr_cache_open**(struct *kr_cache* * *cache*, const struct kr_cdb_api * *api*, struct kr_cdb_opts * *opts*, knot_mm_t * *mm*)
Open/create cache with provided storage options.

Return 0 or an error code

Parameters

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

KR_EXPORT void **kr_cache_close**(struct *kr_cache* * *cache*)
Close persistent cache.

Note This doesn't clear the data, just closes the connection to the database.

Parameters

- `cache`: structure

KR_EXPORT int **kr_cache_sync**(struct *kr_cache* * *cache*)
Run after a row of operations to release transaction/lock if needed.

static bool **kr_cache_is_open**(struct *kr_cache* * *cache*)
Return true if cache is open and enabled.

static void **kr_cache_make_checkpoint**(struct *kr_cache* * *cache*)
(Re)set the time pair to the current values.

KR_EXPORT int **kr_cache_insert_rr**(struct *kr_cache* * *cache*, const knot_rrset_t * *rr*, const knot_rrset_t * *rrsig*, uint8_t *rank*, uint32_t *timestamp*)
Insert RRSet into cache, replacing any existing data.

Return 0 or an errcode

Parameters

- `cache`: cache structure
- `rr`: inserted RRSet
- `rrsig`: RRSIG for inserted RRSet (optional)
- `rank`: rank of the data
- `timestamp`: current time

KR_EXPORT int **kr_cache_clear**(struct *kr_cache* * *cache*)
Clear all items from the cache.

Return 0 or an errcode

Parameters

- `cache`: cache structure

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

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

KR_EXPORT int **kr_cache_materialize**(knot_rdataset_t * *dst*, const struct *kr_cache_p* * *ref*, uint32_t *new_ttl*, knot_mm_t * *pool*)

Variables

const size_t **PKT_SIZE_NOWIRE** = -1
When knot_pkt is passed from cache without ->wire, this is the ->size.

struct kr_cache
#include <api.h> Cache structure, keeps API, instance and metadata.

Public Members

knot_db_t* **db**
Storage instance.

const struct kr_cdb_api* **api**
Storage engine.

uint32_t **hit**
Number of cache hits.

uint32_t **miss**
Number of cache misses.

uint32_t **insert**
Number of insertions.

uint32_t **delete**
Number of deletions.

struct *kr_cache*::@0 **stats**

uint32_t **ttl_min**

uint32_t **ttl_max**
TTL limits.

struct timeval **checkpoint_walltime**
Wall time on the last check-point.

uint64_t **checkpoint_monotime**
Monotonic milliseconds on the last check-point.

struct kr_cache_p

Public Members

uint32_t **time**
The time of inception.

uint32_t **ttl**
TTL at inception moment.

Assuming it fits into int32_t ATM.

uint8_t **rank**
See enum kr_rank.

void* **raw_data**

void * **raw_bound**

struct *kr_cache_p*::@1 *kr_cache_p*::@**2**

4.6.3 Nameservers

Defines

KR_NS_DEAD
See kr_nsrep_update_rtt()

KR_NS_TIMEOUT_RETRY_INTERVAL
If once NS was marked as "timeouted", it won't participate in NS elections at least KR_NS_TIMEOUT_RETRY_INTERVAL milliseconds (now: one minute).

KR_NSREP_MAXADDR

Typedefs

typedef struct *kr_nsrep_rtt_lru_entry* **kr_nsrep_rtt_lru_entry_t**

Enums

enum kr_ns_score
NS RTT score (special values).

Note RTT is measured in milliseconds.

Values:

KR_NS_MAX_SCORE = KR_CONN_RTT_MAX

KR_NS_TIMEOUT = (95 * KR_NS_MAX_SCORE) / 100

KR_NS_LONG = (3 * KR_NS_TIMEOUT) / 4

KR_NS_UNKNOWN = KR_NS_TIMEOUT / 2

KR_NS_PENALTY = 100

KR_NS_GLUED = 10

enum kr_ns_rep
NS QoS flags.

Values:

KR_NS_NOIP4 = 1 << 0
NS has no IPv4.

KR_NS_NOIP6 = 1 << 1
NS has no IPv6.

KR_NS_NOEDNS = 1 << 2
NS has no EDNS support.

enum kr_ns_update_mode
NS RTT update modes.

First update is always KR_NS_RESET unless KR_NS_UPDATE_NORESET mode had choosen.

Values:

KR_NS_UPDATE = 0
Update as smooth over last two measurements.

KR_NS_UPDATE_NORESET
Same as KR_NS_UPDATE, but disable fallback to KR_NS_RESET on newly added entries.

Zero is used as initial value.

KR_NS_RESET
Set to given value.

KR_NS_ADD
Increment current value.

KR_NS_MAX
Set to maximum of current/proposed value.

Functions

typedef **lru_t** (*kr_nsrep_rtt_lru_entry_t*)
NS QoS tracking.

typedef **lru_t** (unsigned)
NS reputation tracking.

KR_EXPORT int **kr_nsrep_set** (struct *kr_query* * *qry*, size_t *index*, const struct sockaddr * *sock*)
Set given NS address.

Return 0 or an error code

Parameters

- `qry`: updated query
- `index`: index of the updated target
- `sock`: socket address to use (sockaddr_in or sockaddr_in6 or NULL)

KR_EXPORT int **kr_nsrep_elect** (struct *kr_query* * *qry*, struct *kr_context* * *ctx*)
Elect best nameserver/address pair from the nsset.

Return 0 or an error code

Parameters

- `qry`: updated query
- `ctx`: resolution context

KR_EXPORT int **kr_nsrep_elect_addr** (struct *kr_query* * *qry*, struct *kr_context* * *ctx*)
Elect best nameserver/address pair from the nsset.

Return 0 or an error code

Parameters

- `qry`: updated query
- `ctx`: resolution context

KR_EXPORT int **kr_nsrep_update_rtt** (struct *kr_nsrep* * *ns*, const struct sockaddr * *addr*, unsigned *score*, kr_nsrep_rtt_lru_t * *cache*, int *umode*)
Update NS address RTT information.

In KR_NS_UPDATE mode reputation is smoothed over last N measurements.

Return 0 on success, error code on failure

Parameters

- `ns`: updated NS representation
- `addr`: chosen address (NULL for first)
- `score`: new score (i.e. RTT), see enum kr_ns_score after two calls with score = KR_NS_DEAD and umode = KR_NS_UPDATE server will be guaranteed to have KR_NS_TIMEOUTED score
- `cache`: RTT LRU cache
- `umode`: update mode (KR_NS_UPDATE or KR_NS_RESET or KR_NS_ADD)

KR_EXPORT int **kr_nsrep_update_rep** (struct *kr_nsrep* * *ns*, unsigned *reputation*, kr_nsrep_lru_t * *cache*)
Update NSSET reputation information.

Return 0 on success, error code on failure

Parameters

- `ns`: updated NS representation
- `reputation`: combined reputation flags, see enum kr_ns_rep
- `cache`: LRU cache

int **kr_nsrep_copy_set**(struct *kr_nsrep* * *dst*, const struct *kr_nsrep* * *src*)
Copy NSSET reputation information and resets score.

Return 0 on success, error code on failure

Parameters

- `dst`: updated NS representation
- `src`: source NS representation

KR_EXPORT int **kr_nsrep_sort**(struct *kr_nsrep* * *ns*, kr_nsrep_rtt_lru_t * *rtt_cache*)
Sort addresses in the query nsrep list.

Return 0 or an error code

Note ns reputation is zeroed, as KR_NS_NOIP{4,6} flags are useless in STUB/FORWARD mode.

Parameters

- `ns`: updated *kr_nsrep*
- `rtt_cache`: RTT LRU cache

struct kr_nsrep_rtt_lru_entry

Public Members

unsigned **score**

uint64_t **tout_timestamp**

struct kr_nsrep
#include <*nsrep.h*> Name server representation.

Contains extra information about the name server, e.g. score or other metadata.

Public Members

unsigned **score**
NS score.

unsigned **reputation**
NS reputation.

const knot_dname_t* **name**
NS name.

struct *kr_context** **ctx**
Resolution context.

union inaddr kr_nsrep::addr[KR_NSREP_MAXADDR]
NS address(es)

Functions

KR_EXPORT int **kr_zonecut_init**(struct *kr_zonecut* * *cut*, const knot_dname_t * *name*, knot_mm_t * *pool*)
Populate root zone cut with SBELT.

Return 0 or error code

Parameters

- `cut`: zone cut
- `name`:
- `pool`:

KR_EXPORT void **kr_zonecut_deinit**(struct *kr_zonecut* * *cut*)
Clear the structure and free the address set.

Parameters

- `cut`: zone cut

KR_EXPORT void **kr_zonecut_set**(struct *kr_zonecut* * *cut*, const knot_dname_t * *name*)
Reset zone cut to given name and clear address list.

Note This clears the address list even if the name doesn't change. TA and DNSKEY don't change.

Parameters

- `cut`: zone cut to be set
- `name`: new zone cut name

KR_EXPORT int **kr_zonecut_copy**(struct *kr_zonecut* * *dst*, const struct *kr_zonecut* * *src*)
Copy zone cut, including all data.

Does not copy keys and trust anchor.

Return 0 or an error code; If it fails with kr_error(ENOMEM), it may be in a half-filled state, but it's safe to deinit...

Note addresses for names in `src` get replaced and others are left as they were.

Parameters

- `dst`: destination zone cut
- `src`: source zone cut

KR_EXPORT int **kr_zonecut_copy_trust**(struct *kr_zonecut* * *dst*, const struct *kr_zonecut* * *src*)
Copy zone trust anchor and keys.

Return 0 or an error code

Parameters

- `dst`: destination zone cut

- `src`: source zone cut

KR_EXPORT int **kr_zonecut_add**(struct *kr_zonecut* * *cut*, const knot_dname_t * *ns*, const knot_rdata_t * *rdata*)

Add address record to the zone cut.

The record will be merged with existing data, it may be either A/AAAA type.

Return 0 or error code

Parameters

- `cut`: zone cut to be populated
- `ns`: nameserver name
- `rdata`: nameserver address (as rdata)

KR_EXPORT int **kr_zonecut_del**(struct *kr_zonecut* * *cut*, const knot_dname_t * *ns*, const knot_rdata_t * *rdata*)

Delete nameserver/address pair from the zone cut.

Return 0 or error code

Parameters

- `cut`:
- `ns`: name server name
- `rdata`: name server address

KR_EXPORT int **kr_zonecut_del_all**(struct *kr_zonecut* * *cut*, const knot_dname_t * *ns*)

Delete all addresses associated with the given name.

Return 0 or error code

Parameters

- `cut`:
- `ns`: name server name

KR_EXPORT KR_PURE pack_t* **kr_zonecut_find**(struct *kr_zonecut* * *cut*, const knot_dname_t * *ns*)

Find nameserver address list in the zone cut.

Note This can be used for membership test, a non-null pack is returned if the nameserver name exists.

Return pack of addresses or NULL

Parameters

- `cut`:
- `ns`: name server name

KR_EXPORT int **kr_zonecut_set_sbelt**(struct *kr_context* * *ctx*, struct *kr_zonecut* * *cut*)

Populate zone cut with a root zone using SBELT :rfc:`1034`

Return 0 or error code

Parameters

- `ctx`: resolution context (to fetch root hints)
- `cut`: zone cut to be populated

KR_EXPORT int **kr_zonecut_find_cached**(struct *kr_context* * *ctx*, struct *kr_zonecut* * *cut*, const knot_dname_t * *name*, const struct *kr_query* * *qry*, bool *restrict *secured*)

Populate zone cut address set from cache.

Return 0 or error code (ENOENT if it doesn't find anything)

Parameters

- `ctx`: resolution context (to fetch data from LRU caches)
- `cut`: zone cut to be populated
- `name`: QNAME to start finding zone cut for
- `qry`: query for timestamp and stale-serving decisions
- `secured`: set to true if want secured zone cut, will return false if it is provably insecure

KR_EXPORT bool **kr_zonecut_is_empty**(struct *kr_zonecut* * *cut*)

Check if any address is present in the zone cut.

Return true/false

Parameters

- `cut`: zone cut to check

struct kr_zonecut

#include <zonecut.h> Current zone cut representation.

Public Members

knot_dname_t* **name**

Zone cut name.

knot_rrset_t* **key**

Zone cut DNSKEY.

knot_rrset_t* **trust_anchor**

Current trust anchor.

struct *kr_zonecut** **parent**

Parent zone cut.

*trie_t** **nsset**

Map of nameserver => address_set (pack_t).

knot_mm_t* **pool**

Memory pool.

4.6.4 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 (f.e. hints)

KR_MODULE_API

Typedefs

typedef uint32_t**(module_api_cb)**(void)

typedef char***(kr_prop_cb)**(void **env*, struct *kr_module* **self*, const char **input*)
Module property callback.

Input and output is passed via a JSON encoded in a string.

Return a free-form JSON output (malloc-ated)

Parameters

- `env`: pointer to the lua engine, i.e. struct engine *env (TODO: explicit type)
- `input`: parameter (NULL if missing/nil on lua level)

Functions

KR_EXPORT int **kr_module_load**(struct *kr_module* * *module*, const char * *name*, const char * *path*)
Load a C module instance into memory.

Return 0 or an error

Parameters

- `module`: module structure
- `name`: module name
- `path`: module search path

KR_EXPORT void **kr_module_unload**(struct *kr_module* * *module*)
Unload module instance.

Parameters

- `module`: module structure

KR_EXPORT const struct *kr_module** **kr_module_embedded**(const char * *name*)
Get embedded module prototype 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.

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.

const kr_layer_api_t***layer)** (struct *kr_module* *self*)
Get a pointer to packet processing API specs.

See docs on that type.

const struct *kr_prop****props)** (void)
Get a pointer to list of properties, terminated by { NULL, NULL, NULL }.

void* **lib**
Shared library handle or RTLD_DEFAULT.

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**

4.6.5 Utilities

Defines

kr_log_info (*fmt*, ...)

kr_log_error (*fmt*, ...)

kr_log_trace_enabled (query)
Return true if the query has request log handler installed.

VERBOSE_STATUS
Block run in verbose mode; optimized when not run.

WITH_VERBOSE(query)

kr_log_verbose

KR_DNAME_GET_STR(dname_str, dname)

KR_RRTYPE_GET_STR(rrtype_str, rrtype)

static_assert(cond, msg)

RDATA_ARR_MAX

kr_rdataset_next(rd)

KR_RRKEY_LEN

SWAP(x, y)
Swap two places.

Note: the parameters need to be without side effects.

Typedefs

typedef void **(* trace_callback_f)**(struct *kr_request* **request*)
Callback for request events.

typedef void **(* trace_log_f)**(const struct *kr_query* **query*, const char **source*, const char **msg*)
Callback for request logging handler.

Functions

KR_EXPORT bool **kr_verbose_set**(bool *status*)
Set verbose mode.

Not available if compiled with -DNOVERBOSELOG.

KR_EXPORT **KR_PRINTF**(1)
Log a message if in verbose mode.

KR_EXPORT **KR_PRINTF**(3)
Log a message through the request log handler.

Return true if the message was logged

Parameters

- `query`: current query
- `source`: message source
- `fmt`: message format

static long **time_diff**(struct timeval * *begin*, struct timeval * *end*)
Return time difference in miliseconds.

Note based on the _BSD_SOURCE timersub() macro

KR_EXPORT char* **kr_strcatdup**(unsigned *n*, ...)
Concatenate N strings.

int **kr_rand_reseed**(void)
Reseed CSPRNG context.

KR_EXPORT uint32_t **kr_rand_uint**(uint32_t *max*)
Get pseudo-random value between zero and max-1 (inclusive).

Passing zero means that any uint32_t should be returned (it's also faster).

KR_EXPORT int **kr_memreserve**(void * *baton*, char ** *mem*, size_t *elm_size*, size_t *want*, size_t * *have*)
Memory reservation routine for knot_mm_t.

KR_EXPORT int **kr_pkt_recycle**(knot_pkt_t * *pkt*)

KR_EXPORT int **kr_pkt_clear_payload**(knot_pkt_t * *pkt*)

KR_EXPORT int **kr_pkt_put**(knot_pkt_t * *pkt*, const knot_dname_t * *name*, uint32_t *ttl*, uint16_t *rclass*, uint16_t *rtype*, const uint8_t * *rdata*, uint16_t *rdlen*)
Construct and put record to packet.

KR_EXPORT void **kr_pkt_make_auth_header**(knot_pkt_t * *pkt*)
Set packet header suitable for authoritative answer.

(for policy module)

KR_EXPORT KR_PURE const char* **kr_inaddr**(const struct sockaddr * *addr*)
Address bytes for given family.

KR_EXPORT KR_PURE int **kr_inaddr_family**(const struct sockaddr * *addr*)
Address family.

KR_EXPORT KR_PURE int **kr_inaddr_len**(const struct sockaddr * *addr*)
Address length for given family, i.e.

sizeof(struct in*_addr).

KR_EXPORT KR_PURE int **kr_sockaddr_len**(const struct sockaddr * *addr*)
Sockaddr length for given family, i.e.

sizeof(struct sockaddr_in*).

KR_EXPORT KR_PURE int **kr_sockaddr_cmp**(const struct sockaddr * *left*, const struct sockaddr * *right*)
Compare two given sockaddr.

return 0 - addresses are equal, error code otherwise.

KR_EXPORT KR_PURE uint16_t **kr_inaddr_port**(const struct sockaddr * *addr*)
Port.

KR_EXPORT int **kr_inaddr_str**(const struct sockaddr * *addr*, char * *buf*, size_t * *buflen*)
String representation for given address as "<addr>#<port>".

KR_EXPORT KR_PURE int **kr_straddr_family**(const char * *addr*)
Return address type for string.

KR_EXPORT KR_CONST int **kr_family_len**(int *family*)
Return address length in given family (struct in*_addr).

KR_EXPORT struct sockaddr* **kr_straddr_socket**(const char * *addr*, int *port*)
Create a sockaddr* from string+port representation (also accepts IPv6 link-local).

KR_EXPORT int **kr_straddr_subnet**(void * *dst*, const char * *addr*)
Parse address and return subnet length (bits).

Warning 'dst' must be at least `sizeof(struct in6_addr)` long.

KR_EXPORT int **kr_straddr_split** (const char * *addr*, char * *buf*, size_t *buflen*, uint16_t * *port*)
Splits ip address specified as "addr@port" or "addr#port" into addr and port and performs validation.

Note if #port part isn't present, then port will be set to 0. buf and port can be set to NULL.

Return kr_error(EINVAL) - addr part doesn't contains valid ip address or #port part is out-of-range (either < 0 either > UINT16_MAX) kr_error(ENOSP) - buflen is too small

KR_EXPORT int **kr_straddr_join** (const char * *addr*, uint16_t *port*, char * *buf*, size_t * *buflen*)
Formats ip address and port in "addr#port" format.

and performs validation.

Note Port always formatted as five-character string with leading zeros.

Return kr_error(EINVAL) - addr or buf is NULL or buflen is 0 or addr doesn't contain a valid ip address kr_error(ENOSP) - buflen is too small

KR_EXPORT KR_PURE int **kr_bitcmp** (const char * *a*, const char * *b*, int *bits*)
Compare memory bitwise.

The semantics is "the same" as for memcmp(). The partial byte is considered with more-significant bits first, so this is e.g. suitable for comparing IP prefixes.

static uint8_t **KEY_FLAG_RANK** (const char * *key*)

static bool **KEY_COVERING_RRSIG** (const char * *key*)

KR_EXPORT int **kr_rrkey** (char * *key*, uint16_t *class*, const knot_dname_t * *owner*, uint16_t *type*, uint16_t *additional*)
Create unique null-terminated string key for RR.

Return key length if successful or an error

Parameters

- `key`: Destination buffer for key size, MUST be KR_RRKEY_LEN or larger.
- `class`: RR class.
- `owner`: RR owner name.
- `type`: RR type.
- `additional`: flags (for instance can be used for storing covered type when RR type is RRSIG).

KR_EXPORT int **kr_ranked_rrarray_add** (ranked_rr_array_t * *array*, const knot_rrset_t * *rr*, uint8_t *rank*, bool *to_wire*, uint32_t *qry_uid*, knot_mm_t * *pool*)

int **kr_ranked_rrarray_set_wire** (ranked_rr_array_t * *array*, bool *to_wire*, uint32_t *qry_uid*, bool *check_dups*, bool(**extraCheck*)(const ranked_rr_array_entry_t *))

KR_PURE char* **kr_pkt_text** (const knot_pkt_t * *pkt*)

KR_PURE char* **kr_rrset_text** (const knot_rrset_t * *rr*)

static *KR_PURE* char* **kr_dname_text** (const knot_dname_t * *name*)

static *KR_CONST* char* **kr_rrtype_text** (const uint16_t *rrtype*)

KR_EXPORT char* **kr_module_call** (struct *kr_context* * *ctx*, const char * *module*, const char * *prop*, const char * *input*)
Call module property.

static uint16_t **kr_rrset_type_maysig** (const knot_rrset_t * *rr*)
Return the (covered) type of an nonempty RRset.

static const char* **lua_push_printf** (lua_State * *L*, const char * *fmt*, ...)
Printf onto the lua stack, avoiding additional copy (thin wrapper).

static char* **kr_straddr** (const struct sockaddr * *addr*)

KR_EXPORT uint64_t **kr_now** ()
The current time in monotonic milliseconds.

Note it may be outdated in case of long callbacks; see uv_now().

int **knot_dname_lf2wire** (knot_dname_t * *dst*, uint8_t *len*, const uint8_t * *lf*)
Convert name from lookup format to wire.

See knot_dname_lf

Note len bytes are read and len+1 are written with *normal* LF, but it's also allowed that the final zero byte is omitted in LF.

Return the number of bytes written (>0) or error code (<0)

static int **kr_dname_lf** (uint8_t * *dst*, const knot_dname_t * *src*, bool *add_wildcard*)
Patched knot_dname_lf.

LF for "." has length zero instead of one, for consistency. (TODO: consistency?)

Note packet is always NULL

Parameters

- `add_wildcard`: append the wildcard label

Variables

KR_EXPORT bool **kr_verbose_status**
Whether in verbose mode.

Only use this for reading.

KR_EXPORT const char* **source**

KR_EXPORT const char const char* **fmt**

const uint8_t **KEY_FLAG_RRSIG** = 0x02

union inaddr
#include <utils.h> Simple storage for IPx address or AF_UNSPEC.

Public Members

struct sockaddr **ip**

struct sockaddr_in **ip4**

struct sockaddr_in6 **ip6**

Defines

KR_EXPORT

KR_CONST

KR_PURE

KR_NORETURN

KR_COLD

KR_PRINTF(n)

uint

kr_ok

kr_strerror(x)

Typedefs

typedef unsigned int **uint**

Functions

static int *KR_COLD* **kr_error**(int *x*)

4.6.6 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.
- *map* - a Crit-bit tree key-value map implementation (public domain) that comes with tests.
- *set* - set abstraction implemented on top of `map` (unused now).
- *pack* - length-prefixed list of objects (i.e. array-list).
- *lru* - LRU-like hash table
- *trie* - a trie-based key-value map, taken from knot-dns

array

A set of simple macros to make working with dynamic arrays easier.

```
MIN(array_push(arr, val), other)
```

Note The C has no generics, so it is implemented mostly using macros. Be aware of that, as direct usage of the macros in the evaluating macros may lead to different expectations:

May evaluate the code twice, leading to unexpected behaviour. This is a price to pay for the absence of proper generics.

Example usage:

```
array_t(const char*) arr;
array_init(arr);

// Reserve memory in advance
if (array_reserve(arr, 2) < 0) {
    return ENOMEM;
}

// Already reserved, cannot fail
array_push(arr, "princess");
array_push(arr, "leia");

// Not reserved, may fail
if (array_push(arr, "han") < 0) {
    return ENOMEM;
}

// It does not hide what it really is
for (size_t i = 0; i < arr.len; ++i) {
    printf("%s\n", arr.at[i]);
}

// Random delete
array_del(arr, 0);
```

Defines

array_t (type)
Declare an array structure.

array_init (array)
Zero-initialize the array.

array_clear (array)
Free and zero-initialize the array (plain malloc/free).

array_clear_mm (array, free, baton)
Make the array empty and free pointed-to memory.

Mempool usage: pass mm_free and a knot_mm_t* .

array_reserve (array, n)
Reserve capacity for at least n elements.

Return 0 if success, <0 on failure

array_reserve_mm (array, n, reserve, baton)
Reserve capacity for at least n elements.

Mempool usage: pass kr_memreserve and a knot_mm_t* .

Return 0 if success, <0 on failure

array_push_mm (array, val, reserve, baton)
Push value at the end of the array, resize it if necessary.

Mempool usage: pass kr_memreserve and a knot_mm_t* .

Note May fail if the capacity is not reserved.

Return element index on success, <0 on failure

array_push(array, val)
Push value at the end of the array, resize it if necessary (plain malloc/free).

Note May fail if the capacity is not reserved.

Return element index on success, <0 on failure

array_pop(array)
Pop value from the end of the array.

array_del(array, i)
Remove value at given index.

Return 0 on success, <0 on failure

array_tail(array)
Return last element of the array.

Warning Undefined if the array is empty.

Functions

static size_t **array_next_count**(size_t *want*)
Simplified Qt containers growth strategy.

static int **array_std_reserve**(void * *baton*, char ** *mem*, size_t *elm_size*, size_t *want*, size_t * *have*)

static void **array_std_free**(void * *baton*, void * *p*)

map

A Crit-bit tree key-value map implementation.

Example usage:

Warning If the user provides a custom allocator, it must return addresses aligned to 2B boundary.

```
map_t map = map_make(NULL);

// Custom allocator (optional)
map.malloc = &mymalloc;
map.baton  = &mymalloc_context;

// Insert k-v pairs
int values = { 42, 53, 64 };
if (map_set(&map, "princess", &values[0]) != 0 ||
    map_set(&map, "prince", &values[1])   != 0 ||
    map_set(&map, "leia", &values[2])     != 0) {
    fail();
}

// Test membership
if (map_contains(&map, "leia")) {
    success();
}
```

(continues on next page)

(continued from previous page)

```
// Prefix search
int i = 0;
int count(const char *k, void *v, void *ext) { (*(int *)ext)++; return 0; }
if (map_walk_prefixed(map, "princ", count, &i) == 0) {
    printf("%d matches\n", i);
}

// Delete
if (map_del(&map, "badkey") != 0) {
    fail(); // No such key
}

// Clear the map
map_clear(&map);
```

Defines

map_walk (*map*, callback, baton)

Functions

static *map_t* **map_make** (struct knot_mm * *pool*)
Creates an new empty critbit map.

Pass NULL for malloc+free.

int **map_contains** (*map_t* * *map*, const char * *str*)
Returns non-zero if map contains str.

void* **map_get** (*map_t* * *map*, const char * *str*)
Returns value if map contains str.

Note: NULL may mean two different things.

int **map_set** (*map_t* * *map*, const char * *str*, void * *val*)
Inserts str into map.

Returns 0 if new, 1 if replaced, or ENOMEM.

int **map_del** (*map_t* * *map*, const char * *str*)
Deletes str from the map, returns 0 on suceess.

void **map_clear** (*map_t* * *map*)
Clears the given map.

int **map_walk_prefixed** (*map_t* * *map*, const char * *prefix*, int(**callback*)(const char *, void *, void *), void * *baton*)
Calls callback for all strings in map with the given prefix.

Returns value immediately if a callback returns nonzero.

Parameters

- `map`:
- `prefix`: required string prefix (empty => all strings)

- `callback`: callback parameters are (key, value, baton)
- `baton`: passed uservalue

struct map_t
#include <map.h> Main data structure.

Public Members

void* **root**

struct knot_mm* **pool**

set

A set abstraction implemented on top of map.

Example usage:

Note The API is based on map.h, see it for more examples.

```
set_t set = set_make(NULL);

// Insert keys
if (set_add(&set, "princess") != 0 ||
    set_add(&set, "prince")   != 0 ||
    set_add(&set, "leia")     != 0) {
    fail();
}

// Test membership
if (set_contains(&set, "leia")) {
    success();
}

// Prefix search
int i = 0;
int count(const char *s, void *n) { (*(int *)n)++; return 0; }
if (set_walk_prefixed(set, "princ", count, &i) == 0) {
    printf("%d matches\n", i);
}

// Delete
if (set_del(&set, "badkey") != 0) {
    fail(); // No such key
}

// Clear the set
set_clear(&set);
```

Defines

set_make
Creates an new, empty critbit set

set_contains (set, str)
Returns non-zero if set contains str

set_add (set, str)
Inserts str into set. Returns 0 if new, 1 if already present, or ENOMEM.

set_del (set, str)
Deletes str from the set, returns 0 on suceess

set_clear (set)
Clears the given set

set_walk (set, callback, baton)
Calls callback for all strings in map

set_walk_prefixed (set, prefix, callback, baton)
Calls callback for all strings in set with the given prefix

Typedefs

typedef *map_t* **set_t**

typedef int **(set_walk_cb)** (const char *, void *)

pack

A length-prefixed list of objects, also an array list.

Each object is prefixed by item length, unlike array this structure permits variable-length data. It is also equivallent to forward-only list backed by an array.

Example usage:

Note Maximum object size is 2^16 bytes, see *pack_objlen_t* If some mistake happens somewhere, the access may end up in an infinite loop. (equality comparison on pointers)

```
pack_t pack;
pack_init(pack);

// Reserve 2 objects, 6 bytes total
pack_reserve(pack, 2, 4 + 2);

// Push 2 objects
pack_obj_push(pack, U8("jedi"), 4)
pack_obj_push(pack, U8("\xbe\xef"), 2);

// Iterate length-value pairs
uint8_t *it = pack_head(pack);
while (it != pack_tail(pack)) {
    uint8_t *val = pack_obj_val(it);
    it = pack_obj_next(it);
}

// Remove object
pack_obj_del(pack, U8("jedi"), 4);

pack_clear(pack);
```

Defines

pack_init (pack)
Zero-initialize the pack.

pack_clear (pack)
Make the pack empty and free pointed-to memory (plain malloc/free).

pack_clear_mm (pack, free, baton)
Make the pack empty and free pointed-to memory.

Mempool usage: pass mm_free and a knot_mm_t* .

pack_reserve (pack, objs_count, objs_len)
Reserve space for *additional* objects in the pack (plain malloc/free).

Return 0 if success, <0 on failure

pack_reserve_mm (pack, objs_count, objs_len, reserve, baton)
Reserve space for *additional* objects in the pack.

Mempool usage: pass kr_memreserve and a knot_mm_t* .

Return 0 if success, <0 on failure

pack_head (pack)
Return pointer to first packed object.

Recommended way to iterate: for (uint8_t *it = *pack_head(pack)*; it != *pack_tail(pack)*; it = pack_obj_next(it))

pack_tail (pack)
Return pack end pointer.

Typedefs

typedef uint16_t **pack_objlen_t**
Packed object length type.

Functions

typedef **array_t** (uint8_t)
Pack is defined as an array of bytes.

static *pack_objlen_t* **pack_obj_len** (uint8_t * *it*)
Return packed object length.

static uint8_t* **pack_obj_val** (uint8_t * *it*)
Return packed object value.

static uint8_t* **pack_obj_next** (uint8_t * *it*)
Return pointer to next packed object.

static uint8_t* **pack_last** (pack_t *pack*)
Return pointer to the last packed object.

static int **pack_obj_push** (pack_t * *pack*, const uint8_t * *obj*, *pack_objlen_t* *len*)
Push object to the end of the pack.

Return 0 on success, negative number on failure

static uint8_t* **pack_obj_find** (pack_t * *pack*, const uint8_t * *obj*, *pack_objlen_t* *len*)
Returns a pointer to packed object.

Return pointer to packed object or NULL

static int **pack_obj_del** (pack_t * *pack*, const uint8_t * *obj*, *pack_objlen_t* *len*)
Delete object from the pack.

Return 0 on success, negative number on failure

static int **pack_clone** (pack_t ** *dst*, const pack_t * *src*, knot_mm_t * *pool*)
Clone a pack, replacing destination pack; (*dst == NULL) is valid input.

Return kr_error(ENOMEM) on allocation failure.

lru

A lossy cache.

Example usage:

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.

```
// Define new LRU type
typedef lru_t(int) lru_int_t;

// Create LRU
lru_int_t *lru;
lru_create(&lru, 5, NULL);

// Insert some values
int *pi = lru_get_new(lru, "luke", strlen("luke"));
if (pi)
    *pi = 42;
pi = lru_get_new(lru, "leia", strlen("leia"));
if (pi)
    *pi = 24;

// Retrieve values
int *ret = lru_get_try(lru, "luke", strlen("luke"));
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]));
    if (val)
        *val = i;
}

// We're done
lru_free(lru);
```

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
- `mm_ctx`: memory context to use for individual key-value pairs, NULL for default

lru_free (table)
Free an LRU created by lru_create (it can be NULL).

lru_reset (table)
Reset an LRU to the empty state (but preserve any settings).

lru_get_try (table, key_, len_)
Find key in the LRU and return pointer to the corresponding value.

Return pointer to data or NULL if not found

Parameters

- `table`: pointer to LRU
- `key_`: lookup key
- `len_`: key length

lru_get_new (table, key_, len_, res)
Return pointer to value, inserting if needed (zeroed).

Return pointer to data or NULL (can be even if memory could be allocated!)

Parameters

- `table`: pointer to LRU
- `key_`: lookup key
- `len_`: key lengthkeys
- `res`: pointer to bool to store result of operation (true if entry is newly added, false otherwise; can be NULL).

lru_apply (table, function, baton)
Apply a function to every item in LRU.

Parameters

- `table`: pointer to LRU

- `function`: enum lru_apply_do (*function)(const char *key, uint len, val_type *val, void *baton) See enum lru_apply_do for the return type meanings.
- `baton`: extra pointer passed to each function invocation

lru_capacity (table)
Return the real capacity - maximum number of keys holdable within.

Parameters

- `table`: pointer to LRU

Enums

enum lru_apply_do
Possible actions to do with an element.

Values:

LRU_APPLY_DO_NOTHING

LRU_APPLY_DO_EVICT

Functions

static *uint* **round_power** (*uint size*, *uint power*)
Round the value up to a multiple of (1 << power).

trie

Typedefs

typedef void* **trie_val_t**
Native API of QP-tries:

- keys are char strings, not necessarily zero-terminated, the structure copies the contents of the passed keys
- values are void* pointers, typically you get an ephemeral pointer to it
- key lengths are limited by 2^32-1 ATM

XXX EDITORS: trie.{h,c} are synced from https://gitlab.labs.nic.cz/knot/knot-dns/tree/68352fc969/src/contrib/qp-trie only with tiny adjustments, mostly #includes and KR_EXPORT.

Element value.

typedef struct trie **trie_t**
Opaque structure holding a QP-trie.

typedef struct trie_it **trie_it_t**
Opaque type for holding a QP-trie iterator.

Functions

KR_EXPORT *trie_t** **trie_create** (knot_mm_t * *mm*)
Create a trie instance.

KR_EXPORT void **trie_free** (*trie_t* * *tbl*)
Free a trie instance.

KR_EXPORT void **trie_clear** (*trie_t* * *tbl*)
Clear a trie instance (make it empty).

KR_EXPORT size_t **trie_weight** (const *trie_t* * *tbl*)
Return the number of keys in the trie.

KR_EXPORT *trie_val_t** **trie_get_try** (*trie_t* * *tbl*, const char * *key*, uint32_t *len*)
Search the trie, returning NULL on failure.

KR_EXPORT *trie_val_t** **trie_get_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.

Return KNOT_EOK for exact match, 1 for previous, KNOT_ENOENT for not-found, or KNOT_E*.

Parameters

- `tbl`: Trie.
- `key`: Searched key.
- `len`: Key length.
- `val`: Must be valid; it will be set to NULL if not found or errored.

int **trie_apply** (*trie_t* * *tbl*, int(**f*)(*trie_val_t* *, void *), void * *d*)
Apply a function to every trie_val_t, in order.

Return First nonzero from f() or zero (i.e. KNOT_EOK).

Parameters

- `d`: Parameter passed as the second argument to f().

KR_EXPORT int **trie_del** (*trie_t* * *tbl*, const char * *key*, uint32_t *len*, *trie_val_t* * *val*)
Remove an item, returning KNOT_EOK if succeeded or KNOT_ENOENT if not found.

If val!=NULL and deletion succeeded, the deleted value is set.

KR_EXPORT *trie_it_t** **trie_it_begin** (*trie_t* * *tbl*)
Create a new iterator pointing to the first element (if any).

KR_EXPORT void **trie_it_next** (*trie_it_t* * *it*)
Advance the iterator to the next element.

Iteration is in ascending lexicographical order. In particular, the empty string would be considered as the very first.

KR_EXPORT bool **trie_it_finished** (*trie_it_t* * *it*)
Test if the iterator has gone past the last element.

KR_EXPORT void **trie_it_free** (*trie_it_t* * *it*)
Free any resources of the iterator. It's OK to call it on NULL.

KR_EXPORT const char* **trie_it_key** (*trie_it_t* * *it*, size_t * *len*)
Return pointer to the key of the current element.

Note The optional len is uint32_t internally but size_t is better for our usage, as it is without an additional type conversion.

KR_EXPORT *trie_val_t** **trie_it_val** (*trie_it_t* * *it*)
Return pointer to the value of the current element (writable).

CHAPTER 5

Modules API reference

- *Supported languages*
- *The anatomy of an extension*
- *Writing a module in Lua*
- *Writing a module in C*
- *Writing a module in Go*
- *Configuring modules*
- *Exposing C/Go module properties*

5.1 Supported languages

Currently modules written in C and LuaJIT are supported. There is also a support for writing modules in Go 1.5+ — the library has no native Go bindings, library is accessible using CGO.

5.2 The anatomy of an extension

A module is a shared object or script defining specific functions, here's an overview.

Note — the *Modules* header documents the module loading and API.

C/Go	Lua	Params	Comment
`X_api()`[1]			API version
`X_init()`	`X.init()`	`module`	Constructor
`X_deinit()`	`X.deinit()`	`module, key`	Destructor
`X_config()`	`X.config()`	`module`	Configuration
`X_layer()`	`X.layer`	`module`	*Module layer*
`X_props()`			List of properties

The `X` corresponds to the module name, if the module name is `hints`, then the prefix for constructor would be `hints_init()`. This doesn't apply for Go, as it for now always implements *main* and requires capitalized first letter in order to export its symbol.

Note: The resolution context `struct kr_context` holds loaded modules for current context. A module can be registered with `kr_context_register()`, which triggers module constructor *immediately* after the load. Module destructor is automatically called when the resolution context closes.

If the module exports a layer implementation, it is automatically discovered by `kr_resolver()` on resolution init and plugged in. The order in which the modules are registered corresponds to the call order of layers.

5.3 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.

Note: The Lua functions retrieve an additional first parameter compared to the C counterparts - a "state". There is no Lua wrapper for C structures used in the resolution context, until they're implemented you can inspect the structures using the ffi library.

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)
```

(continues on next page)

[1] Mandatory symbol.

(continued from previous page)

```
        event.recurrent(conf.interval, function ()
                print(counter.total - counter.last, 'q/s')
                counter.last = counter.total
        end)
end

return counter
```

Tip: The API functions may return an integer value just like in other languages, but they may also return a coroutine that will be continued asynchronously. A good use case for this approach is is a deferred initialization, e.g. loading a chunks of data or waiting for I/O.

```
function counter.init(module)
        counter.total = 0
        counter.last = 0
        counter.failed = 0
        return coroutine.create(function ()
                for line in io.lines('/etc/hosts') do
                        load(module, line)
                        coroutine.yield()
                end
        end)
end
```

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
}
```

There is currently an additional "feature" in comparison to C layer functions: the `consume`, `produce` and `checkout` functions do not get called at all if `state == kres.FAIL` (note that `finish` does get called nevertheless).

Since the modules are like any other Lua modules, you can interact with them through the CLI and and any interface.

Tip: The module can be placed anywhere in the Lua search path, in the working directory or in the MODULESDIR.

5.4 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 API. */
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);
        }

        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.

5.5 Writing a module in Go

The Go modules use CGO to interface C resolver library, there are no native bindings yet. Second issue is that layers are declared as a structure of function pointers, which are not present in Go, the workaround is to declare them in CGO header. Each module must be the `main` package, here's a minimal example:

```
package main

/*
#include "lib/module.h"
*/
import "C"
import "unsafe"

/* Mandatory functions */

//export mymodule_api
func mymodule_api() C.uint32_t {
        return C.KR_MODULE_API
}
func main() {}
```

Warning: Do not forget to prefix function declarations with `//export symbol_name`, as only these will be exported in module.

In order to integrate with query processing, you have to declare a helper function with function pointers to the the layer implementation. Since the code prefacing `import "C"` is expanded in headers, you need the *static inline* trick to avoid multiple declarations. Here's how the preface looks like:

```
/*
#include "lib/layer.h"
#include "lib/module.h"
// Need a forward declaration of the function signature
int finish(kr_layer_t *);
// Workaround for layers composition
static inline const kr_layer_api_t *_layer(void)
{
        static const kr_layer_api_t api = {
                .finish = &finish
        };
        return &api;
}
*/
import "C"
import "unsafe"
```

Now we can add the implementations for the `finish` layer and finalize the module:

```
//export finish
func finish(ctx *C.kr_layer_t) C.int {
        // Since the context is unsafe.Pointer, we need to cast it
        var param *C.struct_kr_request = (*C.struct_kr_request)(ctx.data)
        // Now we can use the C API as well
        fmt.Printf("[go] resolved %d queries\n", C.list_size(&param.rplan.resolved))
        return 0
}

//export mymodule_layer
func mymodule_layer(module *C.struct_kr_module) *C.kr_layer_api_t {
        // Wrapping the inline trampoline function
        return C._layer()
```

(continues on next page)

(continued from previous page)

```
}
```

See the CGO for more information about type conversions and interoperability between the C/Go.

5.5.1 Gotchas

- `main()` function is mandatory in each module, otherwise it won't compile.
- Module layer function implementation must be done in C during `import "C"`, as Go doesn't support pointers to functions.
- The library doesn't have a Go-ified bindings yet, so interacting with it requires CGO shims, namely structure traversal and type conversions (strings, numbers).
- Other modules can be called through C call `C.kr_module_call(kr_context, module_name, module_propery, input)`

5.6 Configuring modules

There is a callback `X_config()` that you can implement, see hints module.

5.7 Exposing C/Go 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 (i.e. *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)
{
```

(continues on next page)

(continued from previous page)

```
        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
```

Note — this relies on function pointers, so the same `static inline` trick as for the `Layer()` is required for C/Go.

5.7.1 Special properties

If the module declares properties `get` or `set`, they can be used in the Lua interpreter as regular tables.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

A

array_next_count (C function), 93
array_std_free (C function), 93
array_std_reserve (C function), 93
array_t (C function), 97

C

cache.backends (C function), 14
cache.clear (C function), 17
cache.close (C function), 15
cache.count (C function), 15
cache.get (C function), 17
cache.max_ttl (C function), 15
cache.min_ttl (C function), 16
cache.ns_tout (C function), 16
cache.open (C function), 15
cache.prune (C function), 17
cache.stats (C function), 15
cache_peek (C function), 76
cache_stash (C function), 76
cookies.config (C function), 48

E

environment variable
    cache.current_size(number), 14
    cache.current_storage(string), 14
    cache.size(number), 14
    cache.storage(string), 14
    env(table), 7
    net.ipv4=true|false, 10
    net.ipv6=true|false, 10
    trust_anchors.hold_down_time=30*day, 12
    trust_anchors.keep_removed=0, 13
    trust_anchors.refresh_time=nil, 12
    worker.count, 20
    worker.id, 20
    worker.pid, 20
event.after (C function), 17
event.cancel (C function), 18
event.recurrent (C function), 18
event.reschedule (C function), 18
event.socket (C function), 18

H

hints.add_hosts (C function), 26
hints.config (C function), 26
hints.del (C function), 27
hints.get (C function), 26
hints.root (C function), 27
hints.root_file (C function), 27
hints.set (C function), 27
hints.use_nodata (C function), 28
hostname (C function), 7

K

KEY_COVERING_RRSIG (C function), 89
KEY_FLAG_RANK (C function), 89
knot_dname_lf2wire (C function), 90
kr_bitcmp (C function), 89
kr_cache_clear (C function), 77
kr_cache_close (C function), 76
kr_cache_insert_rr (C function), 76
kr_cache_is_open (C function), 76
kr_cache_make_checkpoint (C function), 76
kr_cache_materialize (C function), 77
kr_cache_open (C function), 76
kr_cache_peek_exact (C function), 77
kr_cache_sync (C function), 76
kr_cache_ttl (C function), 77
kr_dname_lf (C function), 90
kr_dname_text (C function), 89
kr_error (C function), 91
kr_family_len (C function), 88
kr_inaddr (C function), 88
kr_inaddr_family (C function), 88
kr_inaddr_len (C function), 88
kr_inaddr_port (C function), 88
kr_inaddr_str (C function), 88

kr_memreserve (C function), 88
kr_module_call (C function), 89
kr_module_embedded (C function), 85
kr_module_load (C function), 85
kr_module_unload (C function), 85
kr_now (C function), 90
kr_nsrep_copy_set (C function), 81
kr_nsrep_elect (C function), 80
kr_nsrep_elect_addr (C function), 80
kr_nsrep_set (C function), 80
kr_nsrep_sort (C function), 81
kr_nsrep_update_rep (C function), 80
kr_nsrep_update_rtt (C function), 80
kr_pkt_clear_payload (C function), 88
kr_pkt_make_auth_header (C function), 88
kr_pkt_put (C function), 88
kr_pkt_recycle (C function), 88
kr_pkt_text (C function), 89
KR_PRINTF (C function), 87
kr_qflags_clear (C function), 71
kr_qflags_set (C function), 71
kr_rand_reseed (C function), 88
kr_rand_uint (C function), 88
kr_rank_check (C function), 67
kr_rank_set (C function), 67
kr_rank_test (C function), 67
kr_ranked_rrarray_add (C function), 89
kr_ranked_rrarray_set_wire (C function), 89
kr_resolve_begin (C function), 67
kr_resolve_checkout (C function), 68
kr_resolve_consume (C function), 67
kr_resolve_finish (C function), 68
kr_resolve_plan (C function), 68
kr_resolve_pool (C function), 68
kr_resolve_produce (C function), 67
kr_rplan_deinit (C function), 71
kr_rplan_empty (C function), 71
kr_rplan_find_resolved (C function), 72
kr_rplan_init (C function), 71
kr_rplan_last (C function), 72
kr_rplan_pop (C function), 72
kr_rplan_push (C function), 72
kr_rplan_push_empty (C function), 71
kr_rplan_resolved (C function), 72
kr_rplan_satisfies (C function), 72
kr_rrkey (C function), 89
kr_rrset_text (C function), 89
kr_rrset_type_maysig (C function), 90
kr_rrtype_text (C function), 89
kr_sockaddr_cmp (C function), 88
kr_sockaddr_len (C function), 88
kr_straddr (C function), 90
kr_straddr_family (C function), 88
kr_straddr_join (C function), 89
kr_straddr_socket (C function), 88
kr_straddr_split (C function), 89
kr_straddr_subnet (C function), 88
kr_strcatdup (C function), 87
kr_verbose_set (C function), 87
kr_zonecut_add (C function), 83
kr_zonecut_copy (C function), 82
kr_zonecut_copy_trust (C function), 82
kr_zonecut_deinit (C function), 82
kr_zonecut_del (C function), 83
kr_zonecut_del_all (C function), 83
kr_zonecut_find (C function), 83
kr_zonecut_find_cached (C function), 84
kr_zonecut_init (C function), 82
kr_zonecut_is_empty (C function), 84
kr_zonecut_set (C function), 82
kr_zonecut_set_sbelt (C function), 83

L

lru_t (C function), 79, 80
lua_push_printf (C function), 90

M

map (C function), 19
map_clear (C function), 94
map_contains (C function), 94
map_del (C function), 94
map_get (C function), 94
map_make (C function), 94
map_set (C function), 94
map_walk_prefixed (C function), 94
mode (C function), 7
moduledir (C function), 7
modules.list (C function), 14
modules.load (C function), 14
modules.unload (C function), 14

N

net.bufsize (C function), 11
net.close (C function), 10
net.interfaces (C function), 10
net.list (C function), 10
net.listen (C function), 10
net.outgoing_v4 (C function), 11
net.tcp_pipeline (C function), 11
net.tls (C function), 11
net.tls_padding (C function), 11
net.tls_sticket_secret (C function), 11
net.tls_sticket_secret_file (C function), 12

P

pack_clone (C function), 98
pack_last (C function), 97

pack_obj_del (C function), 98
pack_obj_find (C function), 98
pack_obj_len (C function), 97
pack_obj_next (C function), 97
pack_obj_push (C function), 97
pack_obj_val (C function), 97
package_version (C function), 9
policy.add (C function), 33
policy.del (C function), 34
policy.rpz (C function), 34
policy.suffix_common (C function), 34
policy.todnames (C function), 34
predict.config (C function), 37

R

reorder_RR (C function), 8
resolve (C function), 8
RFC
    RFC 1035, 51
    RFC 3986, 15
    RFC 5011, 12, 21, 49
    RFC 5077, 12
    RFC 6147, 46
    RFC 6761, 35
    RFC 6761#section-6, 26
    RFC 7646, 21
    RFC 7706, 50
    RFC 7858, 31
    RFC 7873, 47
    RFC 8109, 50
    RFC 8145#section-5, 49
    RFC 8198, 50
round_power (C function), 100

S

stats.clear_frequent (C function), 29
stats.frequent (C function), 29
stats.get (C function), 29
stats.list (C function), 29
stats.set (C function), 29
stats.upstreams (C function), 29

T

time_diff (C function), 87
trie_apply (C function), 101
trie_clear (C function), 101
trie_create (C function), 101
trie_del (C function), 101
trie_free (C function), 101
trie_get_ins (C function), 101
trie_get_leq (C function), 101
trie_get_try (C function), 101
trie_it_begin (C function), 101
trie_it_finished (C function), 101
trie_it_free (C function), 101
trie_it_key (C function), 101
trie_it_next (C function), 101
trie_it_val (C function), 102
trie_weight (C function), 101
trust_anchors.add (C function), 13
trust_anchors.add_file (C function), 12
trust_anchors.config (C function), 12
trust_anchors.set_insecure (C function), 13

U

user (C function), 8

V

verbose (C function), 7
view:addr (C function), 35
view:tsig (C function), 36

W

worker.coroutine (C function), 19
worker.sleep (C function), 19
worker.stats (C function), 20