Knot DNS Resolver

Release 1.2.0

CZ.NIC Labs

Apr 25, 2017

Contents

The Knot DNS Resolver is a minimalistic caching resolver implementation. The project provides both a resolver library and a small daemon. Modular architecture of the library keeps the core tiny and efficient, and provides a state-machine like API for extensions.

Building project

Installing from packages

The resolver is packaged for Debian, Fedora, Ubuntu and openSUSE Linux distributions. 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.

Platform considerations

Project	Platforms	Compatibility notes
daemon	UNIX-like[1], Microsoft Windows	C99, libuv provides portable I/O
library	UNIX-like, Microsoft Windows[2]	MSVC not supported, needs MinGW
modules	*varies*	
tests/unit	*equivalent to library*	
tests/integration	UNIX-like	Depends on library injection (see[2])

Requirements

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

---

[1] Known to be running (not exclusively) on FreeBSD, Linux and OS X.

[2] Modules are not supported yet, as the PE/DLL loading is different. Library injection is working with ELF *(or Mach-O flat namespace)* only.

Requirement	Required by	Notes
GNU Make 3.80+	*all*	*(build only)*
pkg-config	*all*	*(build only)*[3]
C compiler	*all*	*(build only)*[4]
libknot 2.1+	*all*	Knot DNS library (requires autotools, GnuTLS and Jansson).
LuaJIT 2.0+	`daemon`	Embedded scripting language.
libuv 1.7+	`daemon`	Multiplatform I/O and services (libuv 1.0 with limitations[5]).

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.
libmemcached	`modules/memcached`	To build memcached backend module.
hiredis	`modules/redis`	To build redis backend module.
Go 1.5+	`modules`	Build modules written in Go.
cmocka	`unit tests`	Unit testing framework.
Doxygen	`documentation`	Generating API documentation.
Sphinx	`documentation`	Building this HTML/PDF documentation.
breathe	`documentation`	Exposing Doxygen API doc to Sphinx.
libsystemd	`daemon`	Systemd socket activation support.

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.

---

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

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

[5] 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** - unknown.
- **NetBSD** - unknown.
- **OpenBSD** - unknown.
- **Mac OS X** - most of the dependencies can be found through Homebrew, with the exception of libknot.

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

Building from sources

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, verbose logging will be disabled as well.

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.

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[6]

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

---

[6] See checksec.sh

Building for packages

The build system supports both DESTDIR and amalgamated builds.

```
$ make install DESTDIR=/tmp/stage # Staged install
$ make all install AMALG=yes # Amalgamated build
```

Amalgamated build assembles everything in one source file and compiles it. It is useful for packages, as the compiler sees the whole program and is able to produce a smaller and faster binary. On the other hand, it complicates debugging.

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

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[7]
daemon	SBINDIR	$(PREFIX)/sbin	
configuration	ETCDIR	$(PREFIX)/etc/kresd	Configuration file, templates.
modules	MODULEDIR	$(LIBDIR)/kdns_modules	[8]
work directory		$(PREFIX)/var/run/ kresd	Run directory for daemon.

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

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.

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.

---

[7] The libkres.pc is installed in $(LIBDIR)/pkgconfig.
[8] Users may install additional modules in ~/.local/lib/kdns_modules or in the rundir of a specific instance.

```
$ 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}
```

Building extras

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

Running unit and integration tests

The unit tests require cmocka and are executed with `make check`.

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.

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`.

Knot DNS Resolver library

Requirements

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

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.

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"*.
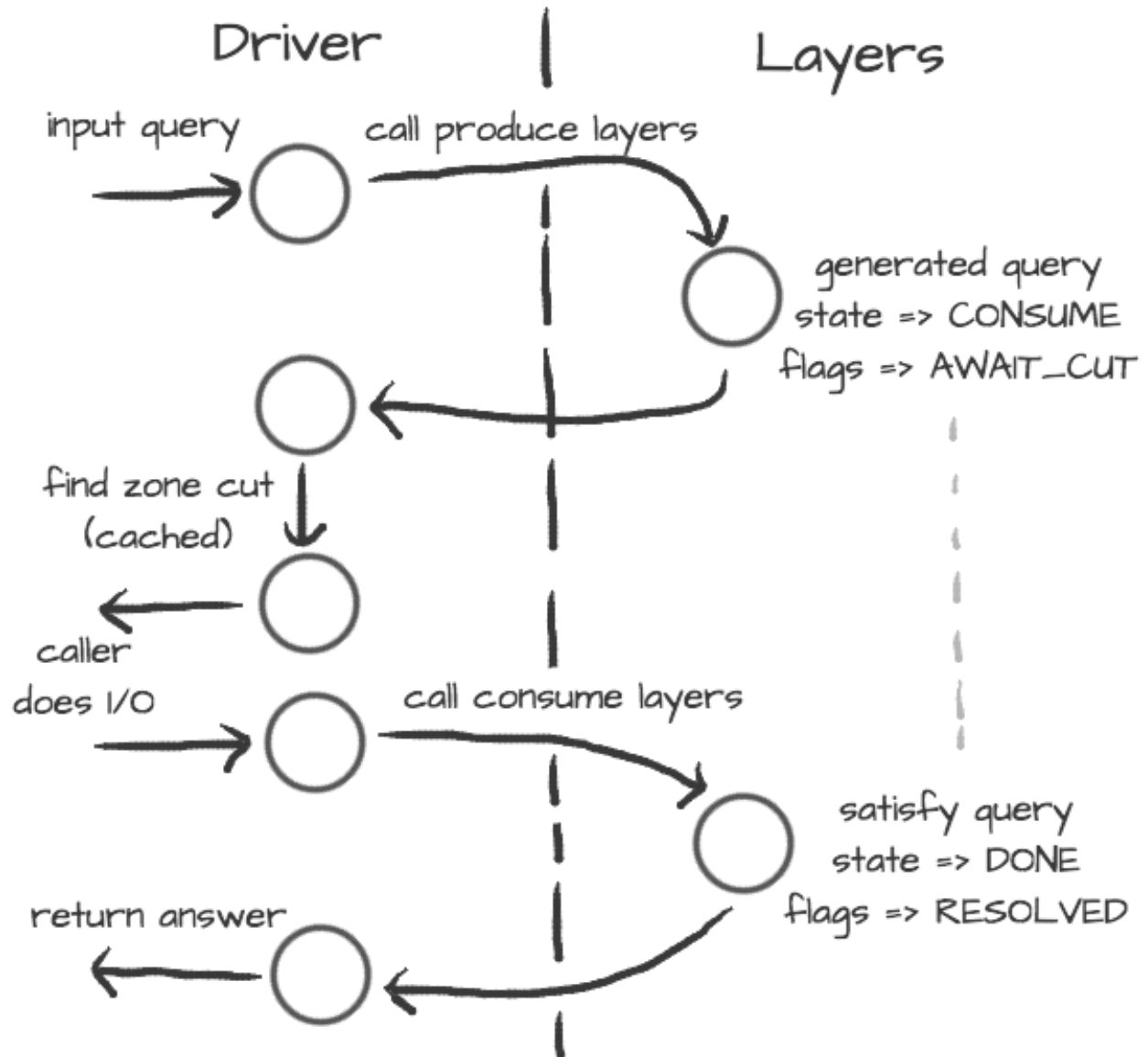
On the other side are *layers*. They are responsible for dissecting the packets and informing the driver about the results. For example, a *produce* layer generates query, a *consume* layer validates answer.

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 `QUERY_AWAIT_CUT` forces driver to fetch zone cut information before the packet is consumed; setting a `QUERY_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
```

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

Writing layers

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 |= QUERY_RESOLVED;
                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;
}
```

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.

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.

Working with domain names

The internal API usually works with domain names in label format, you can convert between text and wire freely.

```lua
local dname = kres.str2dname('business.se')
local strname = kres.dname2str(dname)
```

Working with resource records

Resource records are stored as tables.

```lua
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.

```lua
local rrset = { ... }
local rr = rrset:get(0) -- Return first RR
print(kres.dname2str(rr:owner()))
print(rr:ttl())
print(kres.rr2str(rr))
```

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.

```lua
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')
```

```
        end
end
```

You can enumerate records in the sections.

```lua
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.

```lua
pkt:rcode(kres.rcode.NXDOMAIN)
-- Clear answer and write QUESTION
pkt:clear()
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)
```

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.

```lua
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.

```lua
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.

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

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

API reference

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

Name resolution

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

Example usage of the iterative API:

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

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

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

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

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

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

Typedefs

typedef enum *kr_validation_rank* **kr_validation_rank_t**
Validation rank.

Enums

kr_validation_rank enum
Validation rank.

Values:

- KR_VLDRANK_INITIAL = = 0 -

- KR_VLDRANK_INSECURE = = 1 -

- KR_VLDRANK_BAD = = 2 -

- KR_VLDRANK_MISMATCH = = 3 -

- KR_VLDRANK_UNKNOWN = = 4 -

- KR_VLDRANK_SECURE = = 5 -

Functions

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

    uint32_t **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_lru_t * **cache_rtt**

    kr_nsrep_lru_t * **cache_rep**

    module_array_t * **modules**

    struct kr_cookie_ctx **cookie_ctx**

    kr_cookie_lru_t * **cache_cookie**

    uint32_t **tls_padding**
        See net.tls_padding in ../daemon/README.rst.

    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**
> > Current upstream address.
>
> const struct sockaddr * **dst_addr**
>
> const knot_pkt_t * **packet**
>
> const knot_rrset_t * **opt**
>
> struct kr_request::@3 **qsource**
>
> unsigned **rtt**
> > Current upstream RTT.
>
> struct kr_request::@4 **upstream**
> > Upstream information, valid only in consume() phase.
>
> uint32_t **options**
>
> int **state**
>
> ranked_rr_array_t **answ_selected**
>
> ranked_rr_array_t **auth_selected**
>
> rr_array_t **additional**
>
> bool **answ_validated**
>
> bool **auth_validated**
>
> struct *kr_rplan* **rplan**
>
> int **has_tls**
>
> knot_mm_t **pool**

Defines

QUERY_FLAGS(*X*)
> Log answer with kr_verbose_log(), unless -DNDEBUG.

X(*flag*, *val*)

Enums

kr_query_flag enum
> Query flags.

> *Values:*

Functions

KR_EXPORT KR_CONST const knot_lookup_t * **kr_query_flag_names** (void)
> Query flag names table.

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_next** (struct *kr_query* * *qry*)

> Return query predecessor.

struct **kr_query**

> *#include <rplan.h>* Single query representation.
>
> ### Public Members
>
> struct *kr_query* * **parent**
>
> knot_dname_t * **sname**
>
> uint16_t **stype**
>
> uint16_t **sclass**
>
> uint16_t **id**

uint32_t **flags**

uint32_t **secret**

uint16_t **fails**

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

struct timeval **timestamp**

struct *kr_zonecut* **zone_cut**

struct *kr_nsrep* **ns**

struct kr_layer_pickle * **deferred**

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

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).

Cache

Enums

kr_cache_tag enum
> Cache entry tag.

> *Values:*

>> •KR_CACHE_RR = = 'R' -

>> •KR_CACHE_PKT = = 'P' -

>> •KR_CACHE_SIG = = 'G' -

>> •KR_CACHE_USER = = 0x80 -

kr_cache_rank enum
>   Cache entry rank.

>   Ref: https://tools.ietf.org/html/rfc2181#section-5.4.1

>   **Note**

>>   Be careful about chosen cache rank nominal values.

>>   • AUTH must be > than NONAUTH

>>   • AUTH INSECURE must be > than AUTH (because it attempted validation)

>>   • NONAUTH SECURE must be > than AUTH (because it's valid)

>   *Values:*

>>   •`KR_RANK_BAD = = 0 -`

>>   •`KR_RANK_INSECURE = = 1 -`

>>   •`KR_RANK_EXTRA = = 4 -`

>>   •`KR_RANK_NONAUTH = = 8 -`

>>   •`KR_RANK_AUTH = = 16 -`

>>   •`KR_RANK_SECURE = = 32 -`

kr_cache_flag enum
>   Cache entry flags.

>   *Values:*

>>   •`KR_CACHE_FLAG_NONE = = 0 -`

>>   •`KR_CACHE_FLAG_WCARD_PROOF = = 1 -`

Functions

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 void **kr_cache_sync** (struct *kr_cache* * *cache*)
    Synchronise cache with the backing store.

    Parameters

    - `cache` - structure

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

KR_EXPORT int **kr_cache_peek** (struct *kr_cache* * *cache*, uint8_t *tag*, const knot_dname_t * *name*,
                                uint16_t *type*, struct *kr_cache_entry* ** *entry*, uint32_t * *timestamp*)
    Peek the cache for asset (name, type, tag)

    Note

    The 'drift' is the time passed between the inception time and now (in seconds).

    Return

    0 or an errcode

    Parameters

    - `cache` - cache structure

    - `tag` - asset tag

    - `name` - asset name

    - `type` - asset type

    - `entry` - cache entry, will be set to valid pointer or NULL

    - `timestamp` - current time (will be replaced with drift if successful)

KR_EXPORT int **kr_cache_insert** (struct *kr_cache* * *cache*, uint8_t *tag*, const knot_dname_t * *name*,
                                  uint16_t *type*, struct *kr_cache_entry* * *header*, knot_db_val_t *data*)
    Insert asset into cache, replacing any existing data.

    Return

    0 or an errcode

    Parameters

    - `cache` - cache structure

    - `tag` - asset tag

    - `name` - asset name

    - `type` - asset type

    - `header` - filled entry header (count, ttl and timestamp)

    - `data` - inserted data

KR_EXPORT int **kr_cache_remove** (struct *kr_cache* * *cache*, uint8_t *tag*, const knot_dname_t * *name*,
                                  uint16_t *type*)
    Remove asset from cache.

    Return

    0 or an errcode

Parameters

- `cache` - cache structure

- `tag` - asset tag

- `name` - asset name

- `type` - record type

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_match** (struct *kr_cache* * *cache*, uint8_t *tag*, const knot_dname_t * *name*, knot_db_val_t * *vals*, int *valcnt*)
    Prefix scan on cached items.

Return

number of retrieved keys or an error

Parameters

- `cache` - cache structure

- `tag` - asset tag

- `name` - asset prefix key

- `vals` - array of values to store the result

- `valcnt` - maximum number of retrieved keys

KR_EXPORT int **kr_cache_peek_rank** (struct *kr_cache* * *cache*, uint8_t *tag*, const knot_dname_t * *name*, uint16_t *type*, uint32_t *timestamp*)
    Peek the cache for given key and retrieve it's rank.

Return

rank (0 or positive), or an error (negative number)

Parameters

- `cache` - cache structure

- `tag` - asset tag

- `name` - asset name

- `type` - record type

- `timestamp` - current time

KR_EXPORT int **kr_cache_peek_rr** (struct *kr_cache* * *cache*, knot_rrset_t * *rr*, uint8_t * *rank*, uint8_t * *flags*, uint32_t * *timestamp*)
    Peek the cache for given RRSet (name, type)

Note

> The 'drift' is the time passed between the cache time of the RRSet and now (in seconds).

Return

> 0 or an errcode

Parameters

- `cache` - cache structure

- `rr` - query RRSet (its rdataset may be changed depending on the result)

- `rank` - entry rank will be stored in this variable

- `flags` - entry flags

- `timestamp` - current time (will be replaced with drift if successful)

KR_EXPORT int **kr_cache_materialize** (knot_rrset_t * *dst*, const knot_rrset_t * *src*, uint32_t *drift*, uint *reorder*, knot_mm_t * *mm*)
Clone read-only RRSet and adjust TTLs.

Return

> 0 or an errcode

Parameters

- `dst` - destination for materialized RRSet

- `src` - read-only RRSet (its rdataset may be changed depending on the result)

- `drift` - time passed between cache time and now

- `reorder` - (pseudo)-random seed to reorder the data or zero

- `mm` - memory context

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

Return

> 0 or an errcode

Parameters

- `cache` - cache structure

- `rr` - inserted RRSet

- `rank` - rank of the data

- `flags` - additional flags for the data

- `timestamp` - current time

KR_EXPORT int **kr_cache_peek_rrsig** (struct *kr_cache* * *cache*, knot_rrset_t * *rr*, uint8_t * *rank*, uint8_t * *flags*, uint32_t * *timestamp*)
Peek the cache for the given RRset signature (name, type)

Note

> The RRset type must not be RRSIG but instead it must equal the type covered field of the sought RRSIG.

Return

> 0 or an errcode

Parameters

- `cache` - cache structure
- `rr` - query RRSET (its rdataset and type may be changed depending on the result)
- `rank` - entry rank will be stored in this variable
- `flags` - entry additional flags
- `timestamp` - current time (will be replaced with drift if successful)

KR_EXPORT int **kr_cache_insert_rrsig** (struct *kr_cache* * *cache*, const knot_rrset_t * *rr*, uint8_t
rank, uint8_t *flags*, uint32_t *timestamp*)
Insert the selected RRSIG RRSet of the selected type covered into cache, replacing any existing data.

Note

> The RRSet must contain RRSIGS with only the specified type covered.

Return

> 0 or an errcode

Parameters

- `cache` - cache structure
- `rr` - inserted RRSIG RRSet
- `rank` - rank of the data
- `flags` - additional flags for the data
- `timestamp` - current time

struct **kr_cache_entry**
#include <cache.h> Serialized form of the RRSet with inception timestamp and maximum TTL.

Public Members

uint32_t **timestamp**

uint32_t **ttl**

uint16_t **count**

uint8_t **rank**

uint8_t **flags**

uint8_t **data**[]

struct **kr_cache**
#include <cache.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**
    Maximum TTL of inserted entries.

Nameservers

Defines

KR_NSREP_MAXADDR

kr_nsrep_inaddr(*addr*)

kr_nsrep_inaddr_len(*addr*)

Enums

kr_ns_score enum
    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` -

kr_ns_rep enum
>   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.

kr_ns_update_mode enum
>   NS RTT update modes.

>   *Values:*

>>   •`KR_NS_UPDATE = = 0` - Update as smooth over last two measurements.

>>   •`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** (unsigned)
>   NS reputation/QoS tracking.

KR_EXPORT int **kr_nsrep_set** (struct *kr_query* * *qry*, size_t *index*, uint8_t * *addr*, size_t *addr_len*, int *port*)
>   Set given NS address.

>   **Return**

>>   0 or an error code

>   **Parameters**

>>   • `qry` - updated query

>>   • `index` - index of the updated target

>>   • `addr` - address bytes (struct in_addr or struct in6_addr)

>>   • `addr_len` - address bytes length (type will be derived from this)

>>   • `port` - address port (if <= 0, 53 will be used)

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_lru_t * *cache*, int *umode*)

Update NS address RTT information.

In KR_NS_UPDATE mode reputation is smoothed over last N measurements.

Return

> 0 on success, error code on failure

Parameters

- `ns` - updated NS representation

- `addr` - chosen address (NULL for first)

- `score` - new score (i.e. RTT), see enum kr_ns_score

- `cache` - 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

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.

struct sockaddr **ip**

struct sockaddr_in **ip4**

struct sockaddr_in6 **ip6**

union kr_nsrep::@2 **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

>   ### 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*, uint32_t *timestamp*, 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
>>
>> - `timestamp` - transaction timestamp
>>
>> - `secured` - set to true if want secured zone cut, will return false if it is provably insecure

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.

map_t **nsset**
: Map of nameserver => address_set.

knot_mm_t * **pool**
> Memory pool.

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)

Functions

KR_EXPORT int **kr_module_load** (struct *kr_module* * *module*, const char * *name*, const char * *path*)
> Load 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

struct **kr_prop**
> *#include <module.h>* Module property (named callable).

> A module property has a free-form JSON output (and optional input).

> #### Public Members

> kr_prop_cb * **cb**

> const char * **name**

> const char * **info**

struct **kr_module**
> *#include <module.h>* Module representation.

Public Members

char * **name**
> Name.

module_init_cb * **init**
> Constructor.

module_deinit_cb * **deinit**
> Destructor.

module_config_cb * **config**
> Configuration.

module_layer_cb * **layer**
> Layer getter.

struct *kr_prop* * **props**
> Properties.

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

void * **data**
> Custom data context.

Utilities

Defines

kr_log_info(*fmt*, *...*)

kr_log_error(*fmt*, *...*)

WITH_VERBOSE
> Block run in verbose mode; optimized when not run.

kr_log_verbose

static_assert(*cond*, *msg*)

RDATA_ARR_MAX

kr_rdataset_next(*rd*)

KEY_FLAG_RRSIG

KEY_FLAG_RANK(*key*)

KEY_COVERING_RRSIG(*key*)

KR_RRKEY_LEN

SWAP(*x*, *y*)
> Swap two places.
>
> Note: the parameters need to be without side effects.

Functions

KR_EXPORT bool **kr_verbose_set** (bool *status*)

>   Set verbose mode.

>   Not available if compiled with -DNOVERBOSELOG.

KR_EXPORT void **kr_log_verbose** (const char * *fmt*, ...)

>   Log a message if in verbose mode.

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 unsigned **kr_rand_uint** (unsigned *max*)

>   Get pseudo-random value.

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 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.

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.

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 KR_PURE int **kr_bitcmp** (const char * *a*, const char * *b*, int *bits*)

>   Compare memory bitwise.

KR_EXPORT int **kr_rrkey** (char * *key*, const knot_dname_t * *owner*, uint16_t *type*, uint8_t *rank*)

>   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.
- `owner` - RR owner domain name.
- `type` - RR type.
- `rank` - RR rank (8 bit tag usable for anything).

int **kr_rrmap_add** (*map_t* * *stash*, const knot_rrset_t * *rr*, uint8_t *rank*, knot_mm_t * *pool*)

int **kr_rrarray_add** (rr_array_t * *array*, const knot_rrset_t * *rr*, knot_mm_t * *pool*)

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*)

void **kr_rrset_print** (const knot_rrset_t * *rr*, const char * *prefix*)

void **kr_pkt_print** (knot_pkt_t * *pkt*)

void **kr_dname_print** (const knot_dname_t * *name*, const char * *prefix*, const char * *postfix*)

void **kr_rrtype_print** (const uint16_t *rrtype*, const char * *prefix*, const char * *postfix*)

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

Call module property.

Variables

KR_EXPORT bool **kr_verbose_status**

Whether in verbose mode.

Only use this for reading.

Defines

KR_EXPORT

KR_CONST

KR_PURE

KR_NORETURN

KR_COLD

uint

kr_ok()

kr_strerror(*x*)

Typedefs

typedef unsigned int **uint**

Functions

int **__attribute__** ((__cold__))

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.
- *pack* - length-prefixed list of objects (i.e. array-list).
- *lru* - LRU-like hash table

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:

```c
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.

array_clear_mm(*array*, *free*, *baton*)

array_reserve(*array*, *n*)
   Reserve capacity up to 'n' bytes.

   Return

      0 if success, <0 on failure

array_reserve_mm(*array*, *n*, *reserve*, *baton*)

array_push(*array*, *val*)
   Push value at the end of the array, resize it if necessary.

   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

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

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

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();

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

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

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

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

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

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

Defines

map_walk(*map*, *callback*, *baton*)

Typedefs

typedef void *(* **map_alloc_f**)(void *, size_t)

typedef void(* **map_free_f**)(void *baton, void *ptr)

Functions

map_t **map_make** (void)
> Creates an new, empty critbit map.

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.

int **map_set** (*map_t* * *map*, const char * *str*, void * *val*)
> Inserts str into map, returns 0 on suceess.

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

> **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**

> > map_alloc_f **malloc**

> > map_free_f **free**

> > void * **baton**

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();

// 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 on suceess

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*

```c
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*)
> Free and the pack.

pack_clear_mm(*pack*, *free*, *baton*)

pack_reserve(*pack*, *objs_count*, *objs_len*)
> Incrementally reserve objects in the pack.

pack_reserve_mm(*pack*, *objs_count*, *objs_len*, *reserve*, *baton*)

pack_head(*pack*)
> Return pointer to first packed object.

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.

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

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

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

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

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

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

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_*)
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

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

Parameters

- `table` - pointer to LRU

- `function` - enum lru_apply_do (*function)(const char *key, uint len, val_type *val, void *baton)
See enum lru_apply_do for the return type meanings.

- `baton` - extra pointer passed to each function invocation

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

Parameters

- `table` - pointer to LRU

Enums

lru_apply_do enum
Possible actions to do with an element.

Values:

- `LRU_APPLY_DO_NOTHING` -

- `LRU_APPLY_DO_EVICT` -

Functions

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

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
```

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.keys # File for root keys
[ ta ] bootstrapped root anchor "19036 8 2␣
↪49AAC11D7B6F6446702E54A1607371607A1A41855200FD2CE1CDDE32F24E8FB5"
[ ta ] warning: you SHOULD check the key manually, see: https://data.iana.org/root-
↪anchors/draft-icann-dnssec-trust-anchor.html#sigs
[ ta ] key: 19036 state: Valid
[ ta ] next refresh: 86400000
```

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.

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!

CLI interface

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

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

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
```

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)
                |            -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, you can *use supervisor* that is going to bind to sockets before starting multiple processes.

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.

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. There's a tiny supervisor script for convenience, but you should have a look at real process managers.

```
$ python scripts/supervisor.py ./daemon/kresd -a 127.0.0.1
$ [system] interactive mode
> quit()
> [2016-03-28 16:06:36.795879] process finished, pid = 99342, status = 0, uptime =␣
↪0:00:01.720612
[system] interactive mode
>
```

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

Configuration

In it's simplest form it 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/run/kresd
```

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.

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.

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 *rrcache*. You can enforce specific order by precedence operators **>** and **<**.

```
modules = {
   'hints  > iterate', -- Hints AFTER iterate
   'policy > hints',   -- Policy AFTER hints
   'view   < rrcache'  -- View BEFORE rrcache
}
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
        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
```

Events and services

The 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.

For example a simple web server that doesn't block:

```
local server, headers = require 'http.server', require 'http.headers'
local cqueues = require 'cqueues'
-- Start socket server
local s = server.listen { host = 'localhost', port = 8080 }
assert(s:listen())
-- Compose per-request coroutine
local cq = cqueues.new()
cq:wrap(function()
   s:run(function(stream)
      -- Create response headers
      local headers = headers.new()
      headers:append(':status', '200')
      headers:append('connection', 'close')
      -- Send response and close connection
      assert(stream:write_headers(headers, false))
      assert(stream:write_chunk('OK', true))
      stream:shutdown()
      stream.connection:shutdown()
   end)
   s:close()
end)
-- Hook to socket watcher
event.socket(cq:pollfd(), function (ev, status, events)
   cq:step(0)
end)
```

  • File watchers

Note: Work in progress, come back later!

Configuration reference

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

  • *Environment*

  • *Network configuration*

  • *Trust anchors and DNSSEC*

  • *Modules configuration*

  • *Cache configuration*

  • *Timers and events*

> • *Map over multiple forks*
>
> • *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.

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.

Action	Modes
Use mandatory glue	strict, normal, permissive
Use in-bailiwick glue	normal, permissive
Use any glue records	permissive

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('kresd', 'netgrp')
true
> user('root')
Operation not permitted
```

resolve (qname, qtype[, qclass = *kres.class.IN*, options = *0*, callback = *nil]*)

Parameters

- **qname** (*string*) – Query name (e.g. 'com.')
- **qtype** (*number*) – Query type (e.g. `kres.type.NS`)
- **qclass** (*number*) – Query class *(optional)* (e.g. `kres.class.IN`)
- **options** (*number*) – Resolution options (see query flags)
- **callback** (*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

Example:

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

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
}
[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 (number of outstanding queries that a single client connection can make in parallel). Default is 50.

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

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

`net.tls_padding`([padding])

    Get/set EDNS(0) padding. If set to value >= 2 it will pad the answers to nearest *padding* boundary, e.g. if set to *64*, the answer will have size of multiplies of 64 (64, 128, 192, ...). Setting padding to value < 2 will disable it.

Trust anchors and DNSSEC

`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.config(keyfile)

> **Parameters**
>
> > • **keyfile** (`string`) – File containing DNSKEY records, should be writeable.

You can use only DNSKEY records in managed mode. It is equivalent to CLI parameter `-k <keyfile>` or `trust_anchors.file = keyfile`.

Example output:

```
> trust_anchors.config('root.keys')
[trust_anchors] key: 19036 state: Valid
```

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
```

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 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)

> Get/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()*.
>
> ```
> print(cache.size)
> cache.size = 100 * MB -- equivalent to `cache.open(100 * MB)`
> ```

cache.storage (string)

> Get or change the cache storage backend configuration, see *cache.backends()* for more information. If the new storage configuration is invalid, it is not set.
>
> ```
> print(cache.storage)
> cache.storage = 'lmdb://.'
> ```

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 (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.

cache.min_ttl([ttl])

> **Parameters**

- **ttl** (*number*) – minimum cache TTL (default: 0)

> **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.

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)
```

Map over multiple forks

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.

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
[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*)

pid (number)
> Current worker process PID.

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)
```

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():hasflag(kres.query.DNSSEC_WANT))'
$ echo $?
0
```

Knot DNS Resolver modules

- *Static hints*
- *Statistics collector*
- *Query policies*
- *Views and ACLs*
- *Prefetching records*
- *HTTP/2 services*
- *DNS Application Firewall*
- *Graphite module*
- *Memcached cache storage*
- *Redis cache storage*
- *Etcd module*
- *DNS64*
- *Renumber*
- *DNS Cookies*
- *Version*

Static hints

This is a module providing static hints from `/etc/hosts` like file for forward records (A/AAAA) and reverse records (PTR). You can also use it to change root hints that are used as a safety belt, or if the root NS drops out of cache.

Examples

```
-- Load hints after iterator
modules = { 'hints > iterate' }
-- Load hints before rrcache, custom hosts file
modules = { ['hints < rrcache'] = 'hosts.custom' }
-- Add root hints
hints.root({
   ['j.root-servers.net.'] = { '2001:503:c27::2:30', '192.58.128.30' }
})
-- Set custom hint
hints['localhost'] = '127.0.0.1'
```

Properties

hints.config([path])

> **Parameters**
>
> > - **path** (*string*) – path to hosts file, default: no file
>
> **Returns** { result: bool }

> Clear any configured hints and load specified hosts file. (Root hints are not touched.)

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.

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 (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.] => {
  [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.

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)

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

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.

stats.expiring()

Outputs list of soon-to-expire records as a JSON array. The list maximum size is 5000 entries, make diffs if you want to track it over time.

stats.clear_expiring()

Clear the list of soon expiring records.

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

Query policies

This module can block, rewrite, or alter inbound queries based on user-defined policies. By default, it blocks queries to reverse lookups in private subnets as per **RFC 1918**, **RFC 5735** and **RFC 5737**. You can however extend it to deflect Slow drip DNS attacks for example, or gray-list resolution of misbehaving zones.

There are several policies implemented:

- `pattern` - applies action if QNAME matches regular expression

- `suffix` - applies action if QNAME suffix matches given list of suffixes (useful for "is domain in zone" rules), uses Aho-Corasick string matching algorithm implemented by @jgrahamc (CloudFlare, Inc.) (BSD 3-clause)

- `rpz` - implementes a subset of the RPZ format. Currently it can be used with a zonefile, a binary database support is on the way. Binary database can be updated by an external process on the fly.

- custom filter function

There are several defined actions:

- `PASS` - let the query pass through

- `DENY` - return NXDOMAIN answer

- `DROP` - terminate query resolution, returns SERVFAIL to requestor

- `TC` - set TC=1 if the request came through UDP, forcing client to retry with TCP

- `FORWARD(ip)` - forward query to given IP and proxy back response (stub mode); it can be a single IP (string) or a list of up to four IPs.

- `MIRROR(ip)` - mirror query to given IP and continue solving it (useful for partial snooping)

- `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.

- `TRACE` - pretty-print DNS response packets into the log (useful for debugging weird DNS servers)

> **Warning:** The policy module only looks at the inbound DNS queries. Thus the `FORWARD(ip)` policy does only forward inbound query to the specified IP address(es) and it doesn't and it can't do DNSSEC validation. If you need DNSSEC validation, you either need to disable `FORWARD(ip)` policy or use an upstream DNSSEC-validating resolver.

> **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.

Example configuration

```lua
-- Load default policies
modules = { 'policy' }
-- Whitelist 'www[0-9].badboy.cz'
policy.add(policy.pattern(policy.PASS, '\4www[0-9]\6badboy\2cz'))
-- Block all names below badboy.cz
policy.add(policy.suffix(policy.DENY, {todname('badboy.cz.')}))
-- Custom rule
policy.add(function (req, query)
        if query:qname():find('%d.%d.%d.224\7in-addr\4arpa') then
                return policy.DENY
        end
end)
-- 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 (complete stub mode)
      policy.add(policy.all(policy.FORWARD('2001:DB8::1')))
      -- Print all responses with matching suffix
      policy.add(policy.suffix(policy.TRACE, {todname('rhybar.cz.')}))
-- 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)
```

Properties

policy.PASS
> Pass-through all queries matching the rule.

policy.DENY
> Respond with NXDOMAIN to all queries matching the rule.

policy.DROP
> Drop all queries matching the rule.

policy.TC
> Respond with empty answer with TC bit set (if the query came through UDP).

policy.FORWARD (address)
> Forward query to given IP address.

policy.MIRROR (address)
> Forward query to given IP address.

policy.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'.

policy.TRACE
> Print pretty-formate (dig-like) DNS answers for current query and all its subqueries that Knot Resolver receive from upstream (authoritative) DNS servers. Very useful when dealing with non-compliant DNS servers that violate DNS protocol.

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.all (action)

> **Parameters**
>
> > • **action** – executed action for all queries

Perform action for all queries (no filtering).

policy.pattern (action, pattern)

> **Parameters**
>
> > • **action** – action if the pattern matches QNAME
> >
> > • **pattern** – regular expression

Policy to block queries based on the QNAME regex matching.

policy.suffix (action, suffix_table)

> **Parameters**
>
> > • **action** – action if the pattern matches QNAME
> >
> > • **suffix_table** – table of valid suffixes

Policy to block queries based on the QNAME suffix match.

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[, format])

> **Parameters**
>
> > • **action** – the default action for match in the zone (e.g. RH-value .)
> >
> > • **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
NXDOMAIN	.	**yes**
NODATA	`*.`	*partial*, implemented as NXDOMAIN
Unchanged	`rpz-passthru.`	**yes**
Nothing	`rpz-drop.`	**yes**
Truncated	`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.

```
-- 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' }
```

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:

- subnet - 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.

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')))
```

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.

Prefetching records

The module tracks expiring records (having less than 5% of original TTL) and batches them for predict. 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.

Example configuration

> **Warning:** This module requires 'stats' module to be present and loaded.

```
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.

Exported metrics

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

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

Properties

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

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.

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',
                port = 8053,
                geoip = 'GeoLite2-City.mmdb' -- Optional
        }
}
```

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 = {
        host = 'localhost',
        port = 8053,
        cert = false,
}
```

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

```
http = {
        host = 'localhost',
        port = 8053,
        cert = 'mycert.crt',
        key  = 'mykey.key',
}
```

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

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

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.

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
```

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:

```
http.endpoints['/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}
```

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>'}
```

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
```

```lua
-- Expose the service
http.endpoints['/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'
                end
                value = tonumber(data)
        -- Unsupported method, return 405 Method not allowed
        else
                return 405, 'Cannot do that'
        end
end}
```

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.

```lua
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
```

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.

```lua
http.interface('127.0.0.1', 8080, {
        ['/conf'] = {'application/json', function (h, stream) print('configuration API
↪') end},
        ['/private'] = {'text/html', static_page},
})
```

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

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
    ```

DNS Application Firewall

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

Example configuration

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

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

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

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

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

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

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

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

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

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

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

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

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

Web interface

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

RESTful interface

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

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

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

```
# Get current rule set
$ curl -s -X GET http://localhost: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
```

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.

Example configuration

Only the `host` parameter is mandatory.

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

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

The module supports sending data to multiple servers at once.

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

Dependencies

- luasocket available in LuaRocks

    ```
    $ luarocks install luasocket
    ```

Memcached cache storage

Module providing a cache storage backend for memcached, which makes a good fit for making a shared cache between resolvers.

After loading you can see the storage backend registered and useable.

```
> modules.load 'kmemcached'
> cache.backends()
[memcached://] => true
```

And you can use it right away, see the libmemcached configuration reference for configuration string options, the most essential ones are *–SERVER* or *–SOCKET*. Here's an example for connecting to UNIX socket.

```
> cache.storage = 'memcached://--SOCKET="/var/sock/memcached"'
```

Note: The memcached instance **MUST** support binary protocol, in order to make it work with binary keys. You can pass other options to the configuration string for performance tuning.

Warning: The memcached server is responsible for evicting entries out of cache, the pruning function is not implemented, and neither is aborting write transactions.

Build resolver shared cache

The memcached takes care of the data replication and fail over, you can add multiple servers at once.

```
> cache.storage = 'memcached://--SOCKET="/var/sock/memcached" --SERVER=192.168.1.1 --
↪SERVER=cache2.domain'
```

Dependencies

Depends on the libmemcached library.

Redis cache storage

This modules provides Redis backend for cache storage. Redis is a BSD-license key-value cache and storage server. Like memcached backend, Redis provides master-server replication, but also weak-consistency clustering.

After loading you can see the storage backend registered and useable.

```
> modules.load 'redis'
> cache.backends()
[redis://] => true
```

Redis client support TCP or UNIX sockets.

```
> cache.storage = 'redis://127.0.0.1'
> cache.storage = 'redis://127.0.0.1:6398'
> cache.storage = 'redis:///tmp/redis.sock'
```

It also supports indexed databases if you prefix the configuration string with `DBID@`.

```
> cache.storage = 'redis://9@127.0.0.1'
```

> **Warning:** The Redis client doesn't really support transactions nor pruning. Cache eviction policy shoud be left upon Redis server, see the Using Redis as an LRU cache.

Build distributed cache

See *Redis Cluster* tutorial.

Dependencies

Depends on the hiredis library, which is usually in the packages / ports or you can install it from sources.

Etcd module

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

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

```
$ etcdctl set /kresd/net/127.0.0.1 53
$ etcdctl set /kresd/cache/size 10000000
```

Configures all listening nodes to following configuration:

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

Example configuration

```
modules = {
        ketcd = {
                prefix = '/kresd',
                peer = 'http://127.0.0.1:7001'
        }
}
```

> **Warning:** Work in progress!

Dependencies

- lua-etcd available in LuaRocks

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

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.

> **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.

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')
```

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.

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'}
        }
}
```

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.

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.

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.

Dependencies

- Nettle required for HMAC-SHA256

- development version of libknot (master branch) for DNS cookies handling

Version

Module checks for new version and CVE, and issues warning messages.

Configuration

```
    version.config(2*day)
-- configure period of check (defaults to 1*day)
```

Running

```
modules.load("version")
```

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*

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.

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.

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.

```lua
--- @module Count incoming queries
local counter = {}

function counter.init(module)
        counter.total = 0
        counter.last = 0
        counter.failed = 0
end

function counter.deinit(module)
        print('counted', counter.total, 'queries')
end

-- @function Run the q/s counter with given interval.
function counter.config(conf)
        -- We can use the scripting facilities here
        if counter.ev then event.cancel(counter.ev)
        event.recurrent(conf.interval, function ()
                print(counter.total - counter.last, 'q/s')
                counter.last = counter.total
```

---

[1] Mandatory symbol.

```
        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.

```lua
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*.

```lua
-- 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
}
```

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.

Writing a module in C

As almost all the functions are optional, the minimal module looks like this:

```c
#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*).

```c
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.

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:

```go
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()
}
```

See the CGO for more information about type conversions and interoperability between the C/Go.

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)`

Configuring modules

There is a callback `X_config()` that you can implement, see hints module.

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:

```c
char* get_size(void *env, struct kr_module *m,
               const char *args)
{
        /* Get cache from engine. */
        struct engine *engine = env;
        namedb_t *cache = engine->resolver.cache;

        /* Open read transaction */
        struct kr_cache_txn txn;
        int ret = kr_cache_txn_begin(cache, &txn, NAMEDB_RDONLY);
        if (ret != 0) {
                return NULL;
        }

        /* Read item count */
        char *result = NULL;
        const namedb_api_t *api = kr_cache_storage();
        asprintf(&result, "{ \"result\": %d }", api->count(&txn));
        kr_cache_txn_abort(&txn);

        return result;
}

struct kr_prop *cache_props(void)
{
        static struct kr_prop prop_list[] = {
                /* Callback,    Name,   Description */
                {&get_size, "get_size", "Return number of records."},
                {NULL, NULL, NULL}
        };
```

```
        return prop_list;
}

KR_MODULE_EXPORT(cache)
```

Once you load the module, you can call the module property from the interactive console. *Note* — the JSON output will be transparently converted to Lua tables.

```
$ kresd
...
[system] started in interactive mode, type 'help()'
> modules.load('cached')
> cached.get_size()
[size] => 53
```

Note — this relies on function pointers, so the same `static inline` trick as for the `Layer()` is required for C/Go.

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