A Toolset for the Internet of Threads (IoTh) - Fine-Grained IPv6 Networking in User Space

Renzo Davoli

FOSDEM 2026 - network devroom

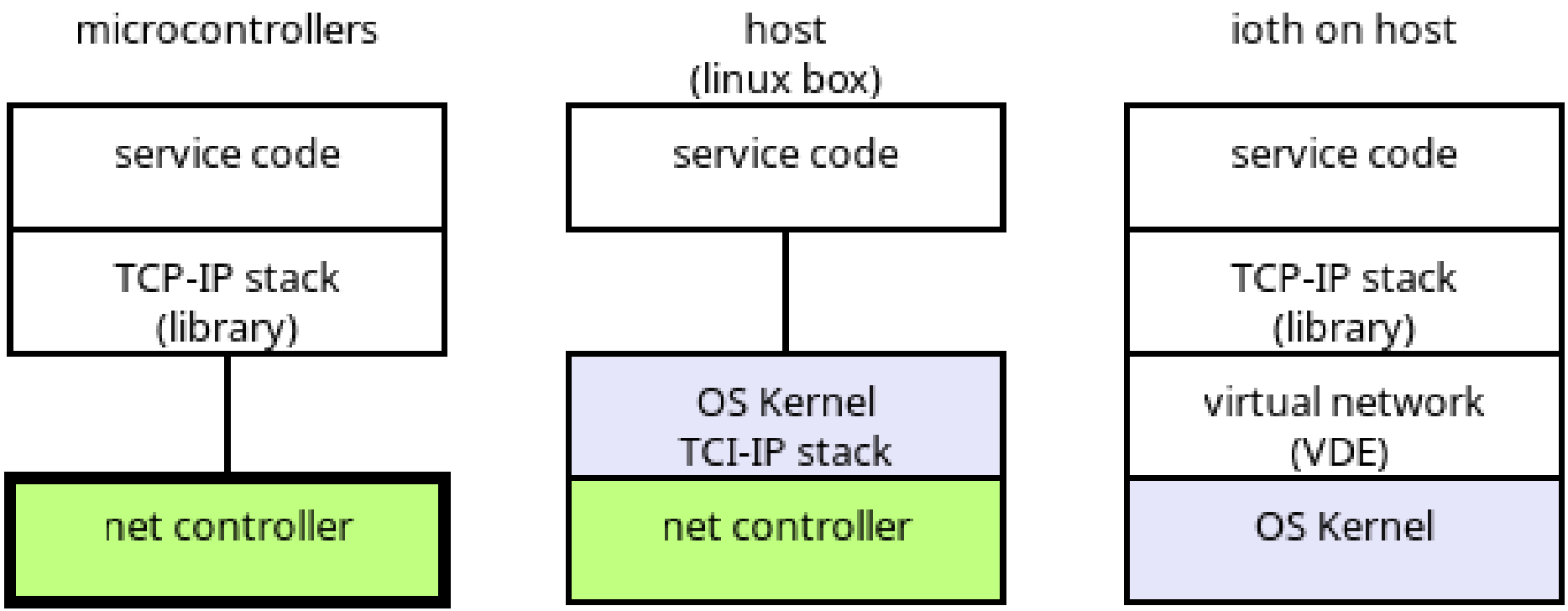
Basic ideas of IoTh

We have plenty of IPv6 addresses available.

TCP and UDP communications need access to services no matters which hardware system or network controller is currently used to provide the service.

IP addresses should be assigned to services (processes).

TCP-IP stacks implemented as a library



TCP-IP stacks implemented as a library: API

Communication: Berkeley sockets

Configuration: ???

- there is not a standard API of TCP-IP stack libraries for embedded systems
- linux: netdevice: based on ioctl obsolete
- linux: AF_NETLINK sockets (NETLINK_ROUTE)

libnlq: Netlink Queue Library (client side)

Networking namespaces, network stacks in user-space need a standard and effective library to configure the stack.

Libnlq is a library:

- giving a simple interface to configure ip addresses and routing
- providing a quick way to set up netlink requests and parse the replies
- which can be used in user-space network stack implementations to support configuration via netlink
- which includes 'drop-in' replacement functions to glibc function still using obsolete netdevice (like if_nametoindex or if_indextoname)
- able to convert many (the most important) ioctl operations defined by netdevice to netlink based calls (this feature can be used both at client and stack side)

libnlq: example

The following source code brings up the interface named vde0 and assigns it IP addresses and routes (both IPv4 and IPv6)

```c
#include <stdint.h>
#include <libnlq.h>

int main(int argc, char *argv[]) {
  uint8_t ipv4addr[] = {192,168,2,2};
  uint8_t ipv4gw[] = {192,168,2,1};
  uint8_t ipv4default[] = {0,0,0,0};
  uint8_t ipv6addr[16] = {0x20, 0x01, 0x07, 0x60, [15] = 0x02};
  uint8_t ipv6gw[16] = {0x20, 0x01, 0x07, 0x60, [15] = 0x01};
  uint8_t ipv6default[16] = {0};

  int ifindex = nlq_if_nametoindex("vde0");

  nlq_linksetupdown(ifindex, 1);
  nlq_ipaddr_add(AF_INET, ipv4addr, 24, ifindex);
  nlq_iproute_add(AF_INET, ipv4default, 0, ipv4gw, 0);
  nlq_ipaddr_add(AF_INET6, ipv6addr, 64, ifindex);
  nlq_iproute_add(AF_INET6, ipv6default, 0, ipv6gw, 0);
}
```

libnlq: Server side usage

Libnlq can be used to support the netlink configuration in user-level stack implementation and libraries.

The core data structure is a table of function pointers (`nlq_request_handlers_table`). NETLINK_ROUTE requests are classified in families: e.g. LINK to configure the interfaces, ADDR the addresses, ROUTE the routing table and so on. A `nlq_request_handlers_table` defines for each family of RT netlink requests a structure of five functions:

- search_entry: it returns the entry matching the parameters and attributes of the request
- get: it returns the details of an entry (or a dump of all the entries);
- new: it creates a new entry;
- del: it must remove an entry;
- set: it updates some values of an entry.

`nlinline`: A quick and clean API for NetLink networking configuring.

NLINLINE (netlink inline) is a library of inline functions providing C programmers with very handy functions to configure network stacks. NLINLINE is entirely implemented in a header file, nlinline.h. Alternative to libnlq clientside only.

- NLINLINE is a simple way to configure networking for network namespaces and Internet of Threads programs.
- NLINLINE does not add dependencies at run-time. It is useful for security critical applications (like PAM modules)
- NLINLINE uses netlink only, it does not depends on the obsolete netdevice (ioctl) API.
- Only the code of referenced inline functions enters in the object and executable code.

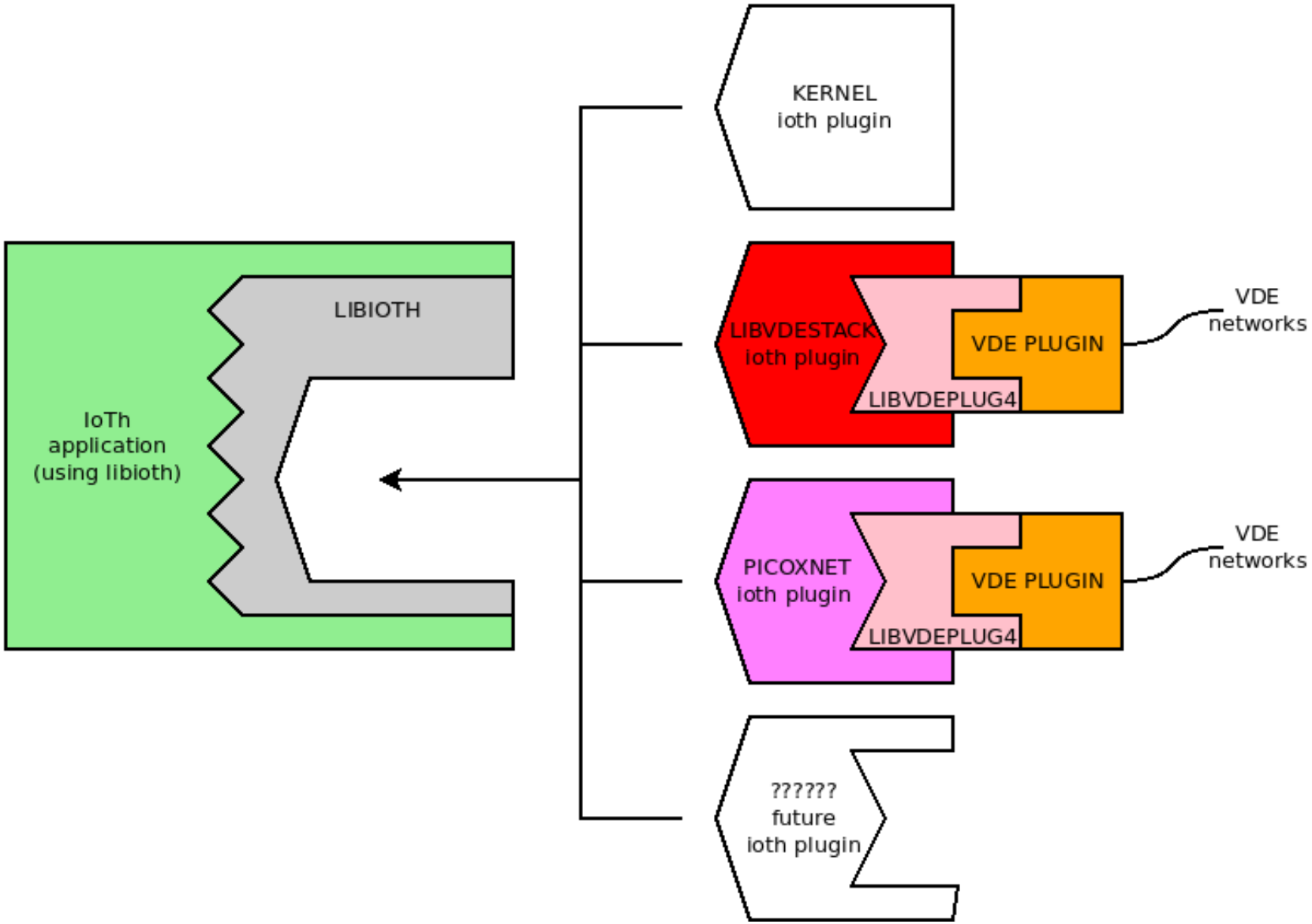
ninline example

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <nlinline.h>

int main(int argc, char *argv[]) {
  uint8_t ipv4addr[] = {192,168,2,2};
  uint8_t ipv4gw[] = {192,168,2,1};
  uint8_t ipv6addr[16] = {0x20, 0x01, 0x07, 0x60, [15] = 0x02};
  uint8_t ipv6gw[16] = {0x20, 0x01, 0x07, 0x60, [15] = 0x01};

  int ifindex = nlinline_if_nametoindex(argv[1]);
  if (ifindex > 0)
    printf("%d\n", ifindex);
  else { perror("nametoindex"); return 1; }

  if (nlinline_linksetupdown(ifindex, 1) < 0) perror("link up");
  if (nlinline_ipaddr_add(AF_INET, ipv4addr, 24, ifindex) < 0) perror("addr ipv4");
  if (nlinline_iproute_add(AF_INET, NULL, 0, ipv4gw, 0) < 0) perror("route ipv4");
  if (nlinline_ipaddr_add(AF_INET6, ipv6addr, 64, ifindex) < 0) perror("addr ipv6")
  if (nlinline_iproute_add(AF_INET6, NULL, 0, ipv6gw, 0) < 0) perror("route ipv6");
  return 0;
}
```

libioth: The unified API for the Internet of Threads

- the API is minimal: Berkeley Sockets + msocket + newstack/delstack.
- the stack implementation can be chosen as a plugin at run time.
- netlink based stack/interface/ip configuration via nlinline.
- ioth sockets are real file descriptors, poll/select/ppoll/pselect/epoll friendly
- plug-ins are loaded in private address namespaces: libioth supports several stacks of the same type (same plugin) even if the stack implementation library was designed to provide just one stack.

libioth idea



libioth API

- newstack There are three flavours of newstack:

```
struct ioth *ioth_newstack(const char *stack, const char *vnl);
struct ioth *ioth_newstackl(const char *stack, const char *vnl, ... /* NULL */);
struct ioth *ioth_newstackv(const char *stack, const char *vnlv[]);
```

- delstack:

```
int ioth_delstack(struct ioth *iothstack);
```

- msocket

```
int ioth_msocket(struct ioth *iothstack, int domain, int type, int protocol);
```

- for everything else... Berkeley Sockets

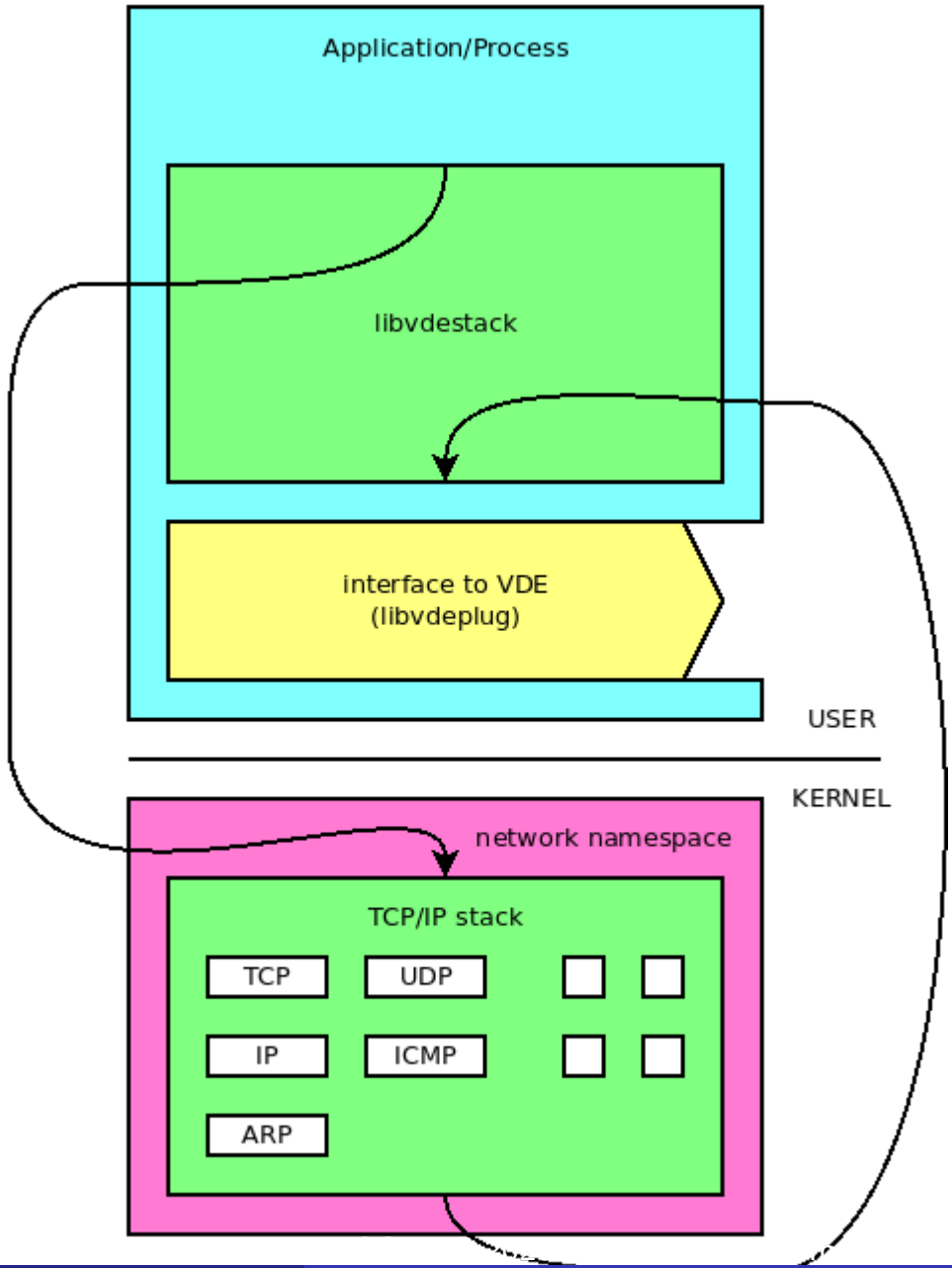
```
ioth_close, ioth_bind, ioth_connect, ioth_listen, ioth_accept, ioth_getsockname,
ioth_getpeername, ioth_setsockopt, ioth_getsockopt, ioth_shutdown, ioth_ioctl,
ioth_fcntl, ioth_read, ioth_readv, ioth_recv, ioth_recvfrom, ioth_recvmsg, ioth_wri
ioth_write, ioth_writev, ioth_send, ioth_sendto and ioth_sendmsg
```

All these function have the same signature and functionnalities of their counterpart without the ioth_ prefix.

libvdestack

libvdestack is a library which uses the Linux kernel implementation of the TCP-IP and makes it available as a user level library.

This is a bit tricky. The library starts a vdestack thread that runs in a separate network namespace. Inside that namespace, a TUN/TAP interface (usually called vde0) is used to forward Layer-2 packets and connect the stack back to VDE in user space. Most of the time, the thread is idle. Only the socket() system call is redirected to the vdestack thread, because once a socket is created, it belongs to the network namespace where it was created.



other stacks

- `Picoxnet`: a stack for the Internet of Threads based on picoTCP-ng + `ioth_picox` The picoxnet module for libioth.
  status: prototype already available on github
- `lwip`:
  status: proof-of-concept work-in-progress

iothdns: Name Resolution support library for the Internet of Threads.

The domain name resolution functions provided by the C library use the TCP-IP stack implemented in the Linux kernel. They are thus unsuitable to support user level implemented stacks like those provided by libioth. Moreover the configuration file pathname is hardcoded: `/etc/resolv.conf`

This library provides support for:

- Client programs that need to query DNS servers
- DNS servers, forwarders, filters that need to parse DNS queries, compose and send back appropriate replies

The library must be initialized.The functions `iothdns_init` and `iothdns_init_strcfg` allow the user to choose the ioth stack to use and to set the configuration parameters using the same syntax of resolv.conf(5). The configuration can be provided as a pathname of a file (`iothdns_init`) or as a string (`iothdns_init_strcfg`).

```
struct iothdns *iothdns_init(struct ioth *stack, char *path_config);
struct iothdns *iothdns_init_strcfg(struct ioth *stack, char *config);
void iothdns_fini(struct iothdns *iothdns);
```

iothdns example queries

iothdns provides some functions to query for IPv4 or IPv6 addresses.

```
int iothdns_lookup_a(struct iothdns *iothdns, const char *name, struct in_addr *a, int n);
int iothdns_lookup_aaaa(struct iothdns *iothdns, const char *name, struct in6_addr *aaaa, int n);
int iothdns_lookup_aaaa_compat(struct iothdns *iothdns, const char *name, struct in6_addr *aaaa, int n);
```

compat retuns IPv6 compat IPv4 addresses, e.g. `::ffff:192.168.1.1`

glibc drop-in replacement

```
int iothdns_getaddrinfo(struct iothdns *iothdns,
    const char *node, const char *service,
    const struct addrinfo *hints,
    struct addrinfo **res);

void iothdns_freeaddrinfo(struct addrinfo *res);

const char *iothdns_gai_strerror(int errcode);

int iothdns_getnameinfo(struct iothdns *iothdns,
    const struct sockaddr *addr, socklen_t addrlen,
    char *host, socklen_t hostlen,
    char *serv, socklen_t servlen, int flags);
```

iothconf: Internet of Threads (IoTh) stack configuration made easy peasy

iothconf configures a TCP/IP stack using configuration directives expressed as character strings. it can use four sources of data:

- static data (IPv4 and/or IPv6)
- DHCP (IPv4, RFC 2131 and 6843)
- router discovery (IPv6, RFC 4861)
- DHCPv6 (IPv6, RFC 8415 and 4704)

The API of the iothconf library has three entries:

```
int ioth_config(struct ioth *stack, char *config);
char *ioth_resolvconf(struct ioth *stack, char *config);
struct ioth *ioth_newstackc(const char *stack_config);
```

- `ioth_config` configures the stack,
- `ioth_resolvconf` return a configuration string for the domain name resolution library (e.g. iothdns). The syntax of the configuration file is consistent with resolv.conf(5).
- `ioth_newstackc` is a shortcut for `ioth_newstack` plus `ioth_config`

Options supported by ioth_config and ioth_newstackc:

```
stack=...:    (ioth_newstackc only) define the ip stack implementation
vnl=...:      (ioth_newstackc only) define the vde network to join
iface=...:    select the interface e.g. iface=eth0 (default value vde0)
ifindex=...:  id of the interface (it can be used instead of iface)
fqdn=...:     set the fully qualified domain name for dhcp,
                dhcpv6 slaac-hash-autoconf
mac=...:      (or macaddr) define the macaddr for eth here below.
              (e.g. eth,mac=10:a1:b2:c3:d4:e5)
eth :         turn on the interface (and set the MAC address if requested
                  or a hash based MAC address if fqdn is defined)
dhcp :        (or dhcp4 or dhcpv4) use dhcp (IPv4)
dhcp6 :       (or dhcpv6) use dhcpv6 (for IPv6)
rd :          (or rd6) use the router discovery protocol (IPv6)
slaac :       use stateless auto-configuration (IPv6) (requires rd)
auto :        shortcut for eth+dhcp+dhcp6+rd
auto4 :       (or autov4) shortcut for eth+dhcp
auto6 :       (or autov6) shortcut for eth+dhcp6+rd
ip=.../.. :   set a static address IPv4 or IPv6 and its prefix length
              example: ip=10.0.0.100/24 or ip=2001:760:1:2::100/64
gw=...:       set a static default route IPv4 or IPv6
dns=...:      set a static address for a DNS server
domain=...:   set a static domain for the dns search
```

iothconf examples:

- IPv4 set static IP and gateway

```
ioth_config(stack, "eth,ip=10.0.0.1/24,gw=10.0.0.254");
```

- IPv4 dhcp

```
ioth_config(stack, "eth,dhcp");
```

- IPv4 and IPv6 using all the available autoconfigurations and set the fully qualified domain name

```
ioth_config(stack, "eth,dhcp,dhcp6,rd,fqdn=host.v2.cs.unibo.it");
```

- the same above using the shortcut auto (= eth,dhcp,dhcp6,rd)

```
ioth_config(stack, "auto,fqdn=host.v2.cs.unibo.it");
```

Create and configure the stack using one call and one configuration string:

```
struct ioth *stack =
    ioth_newstackc("stack=vdestack,vnl=vxvde://234.0.0.1,eth,ip=10.0.0.1/24,gw=10.0.0.254");
struct ioth *stack =
    ioth_newstackc("stack=vdestack,vnl=vxvde://234.0.0.1,eth,dhcp");
struct ioth *stack =
    ioth_newstackc("stack=vdestack,vnl=vxvde://234.0.0.1,auto,fqdn=host.v2.cs.unibo.it");
```

iothconf demo

source code of: `ioth_config_example.c`

```c
#define SPDX_LICENSE "SPDX-License-Identifier: GPL-2.0-or-later"
#include <stdio.h>
#include <ioth.h>
#include <iothconf.h>

int main(int argc, char *argv[1]) {
  ioth_set_license(SPDX_LICENSE);
  struct ioth *stack = ioth_newstack("vdestack", "vde:///tmp/hub");
  ioth_config(stack, "eth,ip=10.0.0.3/24");
  // int fd = ioth_msocket(stack, AF_INET, SOCK_STREAM, 0);
  for(;;)
    pause();
}
```

compile & run

```
gcc -o ioth_config_example  ioth_config_example.c -lioth -liothconf
./ioth_config_example
```

iothconf single string configuration demo

source code of: `ioth_newstackc_example.c`

```c
#define SPDX_LICENSE "SPDX-License-Identifier: GPL-2.0-or-later"
#include <stdio.h>
#include <ioth.h>
#include <iothconf.h>

int main(int argc, char *argv[]) {
        ioth_set_license(SPDX_LICENSE);
        struct ioth *stack = ioth_newstackc(argv[1]);
        // int fd = ioth_msocket(stack, AF_INET, SOCK_STREAM, 0);
        for(;;)
                pause();
}
```

compile & run

```
gcc -o ioth_newstackc_example ioth_newstackc_example.c -lioth -liothconf
./ioth_newstackc_example stack=vdestack,vnl=vde:///tmp/hub,eth,ip=10.0.0.4/24
```

Switching stack implementation is as simple as modify a value of the `stack` option in the string

```
./ioth_newstackc_example stack=picox,vnl=vde:///tmp/hub,eth,ip=10.0.0.4/24
```

iothnamed: a DNS server for the ioth

It supports several features:

- separate TCP-IP stacks for service and forwarding
- access control
- domain delegation (delegated, not delegate)
- static addr definitions
- forwarding
- proxy/caching
- hash based IPv6 addresses
- otip: one time IPv6 addresses

hash based IPv6 addresses

configuration file `localhash+forward.rc`

```
rstack    stack=vdestack,vnl=vde:///tmp/hub
rstack    mac=80:01:01:01:01:01,eth
rstack    ip=192.168.1.24/24
rstack    ip=fc00::24/64
fstack    stack=kernel

dns       8.8.8.8
dns       80.80.80.80

net       local 192.168.1.0/24
net       local fc00::/64
auth      accept local

# define the base address as a static record
auth      static local hash.local
static    AAAA hash.local fc00::

auth      hash local .hash.local hash.local
auth      hrev local hash.local/64

auth      cache local .
auth      fwd local .

option hrevmode always
```

hash based IPv6 addresses

start a vde hub:

```
vde_plug null:// hub:///tmp/hub
```

start iothnamed:

```
iothnamed localhash+forward.rc
```

start the testing namespace and try some queries:

```
vdens -R fc00::24 vde:///tmp/hub
ipconf eth,ip=fc00::1
host renzo.hash.local
  renzo.hash.local has IPv6 address fc00::4cc:8049:6765:d03a
host hicsuntleones.hash.local
  hicsuntleones.hash.local has IPv6 address fc00::9ce7:b35f:d928:ed8a
host fc00::4cc:8049:6765:d03a
  a.3.0.d.5.6.7.6.9.4.0.8.c.c.4.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.c.f.ip6.arpa domain name pointer renzo.hash.local
host prep.ai.mit.edu
  prep.ai.mit.edu is an alias for ftp.gnu.org.
  ftp.gnu.org has address 209.51.188.20
  ftp.gnu.org has IPv6 address 2001:470:142:3::b
```

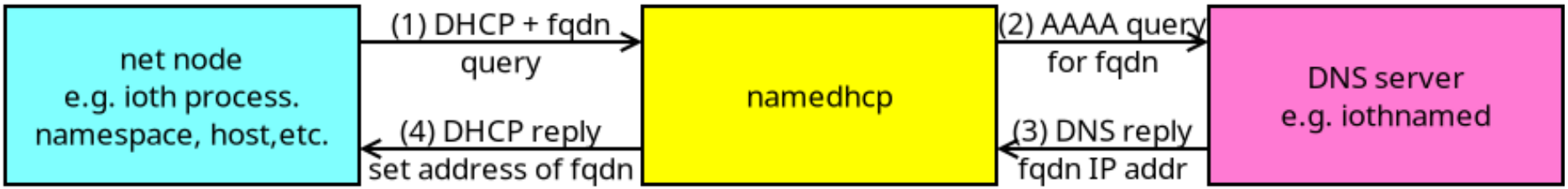
start a namespace and assign a hash based IPv6 addr to it:

```
vdens -R fc00::24 vde:///tmp/hub
ipconf eth,ip=`hashaddr -b fc00::  renzo.hash.local`
```

namedhcp: DIY IP configuration through DHCP

namedhcp is an IPv6 DHCP server implementation for IPV6 stateful autoconfiguration. When namedhcp receives a DHCP query including the fqdn option (option 39 as defined in RFC4704) it queries the DNS for an AAAA record. If there is such a record, the IPv6 address is returned to the DHCP client.

The configuration of the networking nodes (hosts, servers, internet of things objects, internet of threads processes) is very simple in this way, it is sufficient to provide each node with its own fully qualified domain name.



namedhcp: example

start a vde hub:

```
vde_plug null:// hub:///tmp/hub
```

start iothnamed (using the localhash+forward.rc configuration file):

```
iothnamed localhash+forward.rc
```

start a Router Advertisement daemon (if there is not a router)

```
iothradvd -s vde:///tmp/hub -P 10 fc00::/64/L/86400/14400
```

start the namedhcp daemon

```
namedhcp -s "stack=vdestack,vnl=vde:///tmp/hub,ip=fc00::ffff/64,eth" -n fc00::24
```

test the fqdn driven autoconfiguration: node renzo.hash.local

```
vdens -R fc00::24 vde:///tmp/hub
ipconf eth; sleep 1; ipconf dhcpv6,fqdn=renzo.hash.local
```

test the fqdn driven autoconfiguration: node xxxx.hash.local (or any hostname you like).

```
vdens -R fc00::24 vde:///tmp/hub
ipconf eth; sleep 1; ipconf dhcpv6,fqdn=xxxx.hash.local
ping -6 renzo.hash.local
```

OTIP (one time IP addresses)

configuration file for `iothnamed`: `otip+forward.rc`

```
rstack    stack=vdestack,vnl=vde:///tmp/hub
rstack    mac=80:01:01:01:01:01,eth
rstack    ip=192.168.1.24/24
rstack    ip=fc00::24/64
fstack    stack=kernel

dns       8.8.8.8
dns       80.80.80.80

net       local 192.168.1.0/24
net       local fc00::/64
auth      accept local

auth      otip local .otip 2001:760:2e00:ff00:: mypassword

auth      cache local .
auth      fwd local .
```

OTIP (one time IP addresses)

start iothnamed:

```
iothnamed otip+forward.rc
```

start the testing namespace and try some queries:

```
vdens -R fc00::24 vde:///tmp/hub
ipconf eth,ip=fc00::1
host renzo.otip
  renzo.otip has IPv6 address 2001:760:2e00:ff00:1ca6:9bca:b9df:9256
```

. . . wait 32 seconds . . .

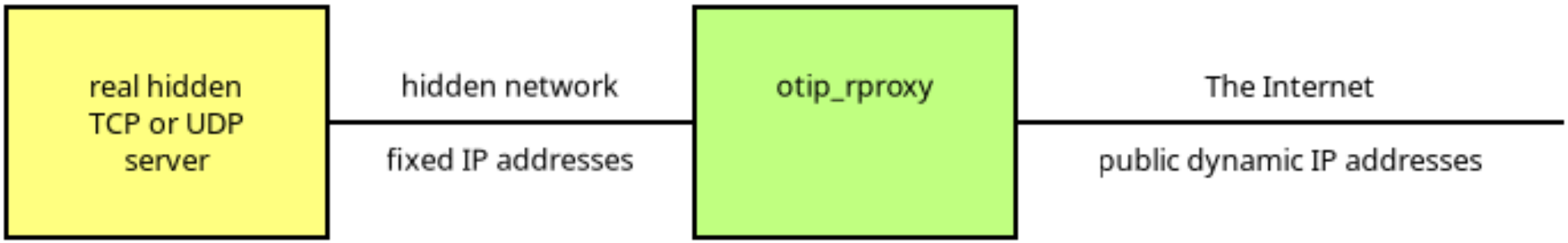
```
host renzo.otip
  renzo.otip has IPv6 address 2001:760:2e00:ff00:c8f2:6482:d4fb:b275
```

use ioth-utils to retrieve the current address:

```
otipaddr -b 2001:760:2e00:ff00:: renzo.otip mypassword
  2001:760:2e00:ff00:c8f2:6482:d4fb:b275
```

otip-utils: otip_rproxy

otip_rproxy is a OTIP enabled reverse proxy. This tool permits to protect TCP or UTP servers using OTIP.



It adds the otip compatibility layer to existing TCP or UDP servers (e.g. web servers, ssh daemons, etc.)

- public dynamic IP address: The address changes each "period" seconds (default value 32)
- otip_rproxy uses several stacks: one for the *internal network* and one for each new dynamic address.
- when the public address expires its stack is usually closed except when there are ongoing TCP connectios.
- In this case the stack is kept alive but all but the listening sockets get closed,
- When the last TCP connection is closed the correspondant stack ends.

thank you!

questions?

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.