hicn Documentation

Release 22.02

Luca Muscariello

Jan 31, 2023

Architecture

Hybrid Information-Centric Networking (hICN) is a network architecture that makes use of IPv6 or IPv4 to implement location-independent communications. It is largely inspired by the pioneer work of Van Jacobson on Content-Centric Networking (RFC 8569, RFC 8609) that is a clean-slate architecture. hICN is based on the Internet protocol and easyier to deploy in today networks and applications. hICN brings many-to-many communications, multi-homing, multi-path, multi-source, group communications to the Internet protocol. The current code implements also transport protocols, with a socket API, for real-time and capacity seeking applications. A scalable stack is available based on VPP and a client stack is provided to support mobile and desktop operating systems.

A detailed description of the architecture is described in the paper

Giovanna Carofiglio, Luca Muscariello, Jordan Augé, Michele Papalini, Mauro Sardara, and Alberto Compagno. 2019. Enabling ICN in the Internet Protocol: Analysis and Evaluation of the Hybrid-ICN Architecture. In Proceedings of the 6th ACM SIGCOMM Conference on Information-Centric Networking (ICN '19). Association for Computing Machinery, New York, NY, USA, 55–66. DOI: https://doi.org/10.1145/3357150.3357394

The project wiki page is full of resources https://wiki.fd.io/view/HICN

CHAPTER 1

Data identifiers and locators

Hybrid ICN makes use of data identifiers to name the data produced by an end host. Data identifiers are encoded using a routable name prefix and a non routable name suffix to provide the ability to index a single IP packet in an prefix is unambigous manner. A full data name is composed of 160 bits. A routable name prefix in IPv4 network is 32 bits long while in IPv6 is 128 bits long. A name prefix is a valid IPv4 or IPv6 address. The 32 rightmost bits are used by the applications to index data within the same stream.

A data source that is using the hicn stack is reacheable through IP routing where a producer socket is listening as the producer name prefix is IP routable.

Locators are IP interface identifiers and are IPv4 or IPv6 addresses. Data consumers are reacheable through IP routing over their locators.

For requests, the name prefix is stored in the destination address field of the IP header while the source address field stored the locator of the consumer.

CHAPTER 2

Producer/Consumer Architecture

Applications make use of the hicn network architecture by using a Prod/Cons API. Each communication socket is connection-less as a data producer makes data available to data consumer by pushing data into a named buffer. Consumers are responsible for pulling data from data producers by sending requests indexing the full data name which index a single MTU sized data packet. The core

CHAPTER 3

Packet forwarding

Packet forwarding leverages IP routing as requests are forwarded using name prefixes and replies using locators.

CHAPTER 4

Relay nodes

A relay node is implemented by using a packet cache which is used to temporarily store requests and replies. The relay node acts as a virtual proxy for the data producers as it caches data packets which can be sent back to data consumer by using the full name as an index. Requests must be cached and forwarded upstream towards data producers which will be able reach back the relay nodes by using the IP locators of the relays. Cached requests store all locators as currently written in the source address field of the request while requests forwarded upstream will get the source address rewritten with the relay node locator. Data packets can reach the original consumers via the relay nodes by using the requence of cached locators.

CHAPTER 5

Code structure

5.1 Introduction

hicn is an open source implementation of Cisco's hICN. It includes a network stack, that implements ICN forwarding path in IPv6, and a transport stack that implements two main transport protocols and a socket API. The transport protocols provide one reliable transport service implementation and a real-time transport service for audio/video media.

5.2 Directory layout

hicn plugin is a VPP plugin that implement hicn packet processing as specified in [1] The transport library is used to implement the hicn host stack and makes use of libmemif as high performance connector between transport and the network stack. The transport library makes use of VPP binary API to configure the local namespace (local face management).

5.3 Release note

The current master branch provides the latest release which is compatible with the latest VPP stable. No other VPP releases are supported nor maintained. At every new VPP release distribution hicn master branch is updated to work with the latest stable release. All previous stable releases are discontinued and not maintained. The user who is interested in a specific release can always checkout the right code tree by searching the latest commit under a given git tag carrying the release version.

The Hybrid ICN software distribution can be installed for several platforms. The network stack comes in two different implementations: one scalable based on VPP and one portable based on IPC and sockets.

The transport stack is a unique library that is used for both the scalable and portable network stacks.

5.4 Supported platforms

- Ubuntu 20.04 LTS (amd64, arm64)
- Android 10 (amd64, arm64)
- iOS 15
- macOS 12.3
- Windows 10

Other platforms and architectures may work. You can either use released packages, or compile hicn from sources.

5.4.1 Ubuntu

```
curl -s https://packagecloud.io/install/repositories/fdio/release/script.deb.sh |␣
↪sudo bash
```

The following debian packages for Ubuntu are available dor amd64 and arm64

```
facemgr-dev
facemgr
hicn-apps-dev
hicn-apps
hicn-light
hicn-plugin-dev
hicn-plugin
libhicn-dev
libhicn
libhicnctrl-dev
libhicnctrl-modules
libhicnctrl
libhicntransport-dev
libhicntransport-io-modules
libhicntransport
```

5.4.2 macOS

```
brew tap icn-team/hicn-tap
brew install hicn
```

or

```
git clone https://github.com/FDio/hicn.git
$ cd hicn
$ OPENSSL_ROOT_DIR=/usr/local/opt/openssl\@1.1 make build-release
```

5.4.3 Android

hICN is built as a native library for the Android NDK which are packaged as Android archives AAR and made available in a Maven repository in Github Packages in

https://github.com/orgs/icn-team/packages

To build from sources, refer to the Android SDK in

https://github.com/icn-team/android-sdk

Install the applications via the Google Play Store

https://play.google.com/store/apps/developer?id=ICN+Team

5.4.4 iOS

Clone this distro

```
git clone https://github.com/icn-team/ios-sdk.git
cd ios-sdk
```

Compile everything (dependencies and hICN modules)

```
make update
make all
```

Compile everything with Qt (dependencies, hICN modules and Viper dependencies)

```
make update
make all_qt
```

5.4.5 Windows

Install vcpkg

```
git clone https://github.com/icn-team/windows-sdk
.\windows-sdk\scripts\init.bat
```

```
cd windows-sdk
make all
```

5.4.6 Docker

Several docker images are nightly built with the latest software for Ubuntu 18 LTS (amd64/arm64), and available on docker hub at https://hub.docker.com/u/icnteam.

The following images are nightly built and maintained.

```
docker pull icnteam/vswitch:amd64
docker pull icnteam/vswitch:arm64

docker pull icnteam/vserver:amd64
docker pull icnteam/vserver:arm64

docker pull icnteam/vhttpproxy:amd64
docker pull icnteam/vhttpproxy:arm64
```

Other Dockerfiles are included in the main git repo for development.

5.4.7 Vagrant

Vagrant boxes for a virtual switch are available at https://app.vagrantup.com/icnteam

```
vagrant box add icnteam/vswitch
```

Supported providers are libvirt, vmware and virtualbox.

5.5 References

Giovanna Carofiglio, Luca Muscariello, Jordan Augé, Michele Papalini, Mauro Sardara, and Alberto Compagno. 2019. Enabling ICN in the Internet Protocol: Analysis and Evaluation of the Hybrid-ICN Architecture. In Proceedings of the 6th ACM Conference on Information-Centric Networking (ICN '19). Association for Computing Machinery, New York, NY, USA, 55–66. DOI: https://doi.org/10.1145/3357150.3357394

5.6 License

This software is distributed under the following license:

CHAPTER 6

Core library

6.1 Introduction

libhicn provides a support library coded in C designed to help developers embed Hybrid ICN (hICN) functionalities in their applications (eg. forwarder, socket API, etc.). Its purpose is to follow the hICN specification for which it provides a reference implementation, abstracting the user from all internal mechanisms, and offering an API independent of the packet format (eg. IPv4 or IPv6). The library is designed to be portable across both desktop and mobile platforms, and we currently aim at supporting Linux, Android, OSX and iOS, by writing the necessary adapters to realize hICN functionality in userspace according to the available APIs and permissions that each system offers.

The library consists in several layers:

- the core library (hicn.h) provides a standard hICN packet format, as well as an API allowing manipulation of packet headers;
- an hICN helper, allowing an hICN stack to be built in userspace in a portable way, based on TUN devices and accessible though file descriptors;
- a network layer allow the sending an receiving of hICN packets on those file descriptors, implementing both source and destination address translation as required by the hICN mechanisms;
- finally, a "transport" API allows the forging of dummy interest and data packets.

A commandline interface (hicnc) is also provided that uses the library and can for instance be used as a test traffic generator. This interface can be run as either a consumer, a producer, or a simple forwarder.

6.2 Directory layout

```
.
+-- CMakeLists.txt          CMkake global build file
+-- doc                     Package documentation
+-- README.md               This file
+-- src
```

(continues on next page)

(continued from previous page)

```
|    +-- base.h                  Base definitions for hICN implementation
|    +-- CMakeLists.txt          CMake library build file
|    +-- common.{h,c}            Harmonization layer across supported platforms
|    +-- compat.{h,c}            Compatibility layer for former API
|    +-- error.{h,c}             Error management files
|    +-- header.h                hICN header definitions
|    +-- hicn.h                  Master include file
|    +-- mapme.{h,c}             MAP-Me : anchorless producer mobility mechanisms
|    +-- name.{h,c}              hICN naming conventions and name processing + IP helpers
|    +-- ops.{h,c}               Protocol-independent hICN operations
|    +-- protocol/*              Protocol headers + protocol-dependent implementations
|    +-- protocol.h              Common file for protocols
```

6.3 Using libhicn

6.3.1 Dependencies

Build dependencies:

- C11 (clang / gcc)
- CMake 3.4

Basic dependencies: None

6.4 Installation

6.4.1 Release mode

```
mkdir build
cd build
cmake ..
make
sudo make install
```

6.4.2 Debug mode

```
mkdir debug
cd debug
cmake .. -DCMAKE_BUILD_TYPE=Debug
make
sudo make install
```

CHAPTER 7

VPP Plugin

7.1 Introduction

A high-performance Hybrid ICN forwarder as a plugin to VPP.

The plugin provides the following functionalities:

- Fast packet processing
- Interest aggregation
- Content caching
- Forwarding strategies

7.2 Quick start

All of these commands should be run from the code tree root.

VPP installed with DEB pkg:

```
cd hicn-plugin
mkdir -p build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX=/usr
make
sudo cmake --build . -- install
```

VPP source code - build type `RELEASE`:

```
cd hicn-plugin
mkdir -p build
cd build
cmake .. -DVPP_HOME=<vpp dir>/build-root/install-vpp-native/vpp -DCMAKE_INSTALL_
↪PREFIX=<vpp src>/build-root/install-vpp-native/vpp
```

(continues on next page)

(continued from previous page)

```
make
cmake --build . -- install
```

VPP source code - build type `DEBUG`:

```
cd hicn-plugin
mkdir -p build
cd build
cmake .. -DCMAKE_BUILD_TYPE=DEBUG -DVPP_HOME=<vpp dir>/build-root/install-vpp_debug-
↪native/vpp -DCMAKE_INSTALL_PREFIX=<vpp src>/build-root/install-vpp_debug-native/vpp
make
cmake --build . -- install
```

CMAKE variables:

- `CMAKE_INSTALL_PREFIX`: set the install directory for the hicn-plugin. This is the common path to the lib folder containing vpp_plugins and vpp_api_test_plugins folders. Default is /usr/local.
- `VPP_HOME`: set the directory containing the include and lib directories of vpp.

7.3 Using hICN plugin

7.3.1 Dependencies

Build dependencies:

- VPP 22.02
  - DEB packages (can be found https://packagecloud.io/fdio/release/install):
    - vpp
    - libvppinfra-dev
    - vpp-dev

Runtime dependencies:

- VPP 22.02
  - DEB packages (can be found https://packagecloud.io/fdio/release/install):
    - vpp
    - vpp-plugin-core
    - vpp-plugin-dpdk (optional - only to use DPDK compatible nics)

Hardware support (not mandatory):

- DPDK compatible NICs

7.4 Getting started

In order to start, the hICN plugin requires a running instance of VPP. The steps required to successfully start hICN are:

- Setup the host to run VPP

- Configure VPP to use DPDK compatible nics
- Start VPP
- Configure VPP interfaces
- Configure and start hICN

Detailed information for configuring VPP can be found at https://wiki.fd.io/view/VPP.

7.4.1 Setup the host for VPP

It is preferable to have hugepages enabled in the system, although it is not a requirement:

```
sudo sysctl -w vm.nr_hugepages=1024
```

In order to use a DPDK interface, the `uio` and `uio_pci_generic` or `vfio_pci` modules need to be loaded in the kernel.

```
sudo modprobe uio
sudo modprobe uio_pci_generic
sudo modprobe vfio_pci
```

If the DPDK interface we want to assign to VPP is up, we must bring it down:

```
sudo ifconfig <interface_name> down
```

or

```
sudo ip link set <interface_name> down
```

7.4.2 Configure VPP

The file `/etc/VPP/startup.conf` contains a set of parameters to setup VPP at startup. The following example sets up VPP to use a DPDK interface:

```
unix {
  nodaemon
  log /tmp/vpp.log
  full-coredump
}

api-trace {
  on
}

api-segment {
  gid vpp
}

dpdk {
  dev 0000:08:00.0
}

plugins {
        ## Disable all plugins by default and then selectively enable specific plugins
```

(continues on next page)

(continued from previous page)

```
        plugin default { disable }
        plugin dpdk_plugin.so { enable }
        plugin acl_plugin.so { enable }
        plugin memif_plugin.so { enable }
        plugin hicn_plugin.so { enable }

        ## Enable all plugins by default and then selectively disable specific plugins
        # plugin dpdk_plugin.so { disable }
        # plugin acl_plugin.so { disable }
}
```

`0000:08:00.0` must be replaced with the actual PCI address of the DPDK interface.

7.4.3 Start VPP

VPP can be started as a process or a service:

Start VPP as a service in Ubuntu 20.04+:

```
sudo systemctl start vpp
```

Start VPP as a process:

```
sudo vpp -c /etc/vpp/startup.conf
```

7.4.4 Configure hICN plugin

The hICN plugin can be configured either using the VPP command-line interface (CLI), through a configuration file or through the VPP binary API.

hICN plugin CLI

The CLI commands for the hICN plugin start all with the `hicn` keyword. To see the full list of command available type:

```
sudo vppctl
vpp# hicn ?
```

`hicn face show`: list the available faces in the forwarder.

```
hicn face show [<face_id>| type <ip/udp>]
  <face_id>                    :face id of which we want to display the informations
  <ip/udp>                     :shows all the ip or udp faces available
```

`hicn pgen client`: set an vpp forwarder as an hicn packet generator client.

```
hicn pgen client src <addr> n_ifaces <n_ifaces> name <prefix> lifetime <interest-
↪lifetime> intfc <data in-interface> max_seq <max sequence number> n_flows <number␣
↪of flows>
  <src_addr>                  :source address to use in the interests, i.e., the␣
↪locator for routing the data packet back
  <n_ifaces>                  :set the number of ifaces (consumer faces) to emulate. If␣
↪more than one, each interest is sent <n_ifaces> times, each of it with a different␣
↪source address calculated from <src_addr>
```

(continues on next page)

(continued from previous page)

```
  <prefix>                    :prefix to use to generate hICN names
  <interest-lifetime>         :lifetime of the interests
  <data in-interface>         :interface through which the forwarder receives data
  <max sequence number>       :max the sequence number to use in the interest. Cycling␣
↪between 0 and this value
  <number of flows>           :emulate multiple flows downloaded in parallel
```

`hicn pgen server`: set an vpp forwarder as an hicn packet generator client.

```
hicn pgen server name <prefix> intfc <interest in-interface> size <payload_size>
  <prefix>                        :prefix to use to reply to interest
  <interest in-interface>         :interface through which the forwarder receives␣
↪interest
  <payload_size>                  :size of the data payload
```

`hicn show`: show forwarder information.

```
hicn show [detail] [strategies]
  <detail>                        :shows additional details as pit,cs entries␣
↪allocation/deallocation
  <strategies>                    :shows only the available strategies int he forwarder
```

`hicn strategy mw set`: set the weight for a face.

```
hicn strategy mw set prefix <prefix> face <face_id> weight <weight>
  <prefix>                        :prefix to which the strategy applies
  <face_id>                       :id of the face to set the weight
  <weight>                         :weight
```

`hicn enable`: enable hICN forwarding pipeline for an ip prefix.

```
hicn enable <prefix>
  <prefix>                        :prefix for which the hICN forwarding pipeline is␣
↪enabled
```

`hicn disable`: disable hICN forwarding pipeline for an ip prefix.

```
hicn enable <prefix>
  <prefix>                        :prefix for which the hICN forwarding pipeline is␣
↪disable
```

hICN plugin configuration file

A configuration can be use to setup the hicn plugin when vpp starts. The configuration file is made of a list of CLI commands. In order to set vpp to read the configuration file, the file `/etc/vpp/startup.conf` needs to be modified as follows:

```
unix {
  nodaemon
  log /tmp/vpp.log
  full-coredump
  startup-config <path to configuration file>
}
```

hICN plugin binary API

The binary api, or the vapi, can be used as well to configure the hicn plugin. For each CLI command there is a corresponding message in the binary api. The list of messages is available in the file hicn.api (located in `hicn/hicn-plugin/src/`).

7.4.5 Example: consumer and producer ping

In this example, we connect two vpp forwarders, A and B, each of them running the hicn plugin. On top of forwarder A we run the `ping_client` application, on top of forwarder B we run the `ping_server` application. Each application connects to the underlying forwarder through a memif-interface. The two forwarders are connected through a dpdk link.

Forwarder A (client)

```
sudo vppctl
vpp# set interface ip address TenGigabitEtherneta/0/0 2001::2/64
vpp# set interface state TenGigabitEtherneta/0/0 up
vpp# ip route add b002::1/64 via remote 2001::3 TenGigabitEtherneta/0/0
vpp# hicn enable b002::1/64
```

Forwarder B (server)

```
sudo vppctl
vpp# set interface ip address TenGigabitEtherneta/0/1 2001::3/64
vpp# set interface state TenGigabitEtherneta/0/1 up
```

Once the two forwarder are started, run the `hicn-ping-server` application on the host where the forwarder B is running:

```
sudo hicn-ping-server -n b002::1/128 -z memif_module
```

Then `hicn-ping-client` on the host where forwarder B is running:

```
sudo hicn-ping-client -n b002::1 -z memif_module
```

7.4.6 Example: packet generator

The packet generator can be used to test the performance of the hICN plugin, as well as a tool to inject packet in a forwarder or network for other test use cases It is made of two entities, a client that inject interest into a vpp forwarder and a server that replies to any interest with the corresponding data. Both client and server can run on a vpp that is configured to forward interest and data as if they were regular ip packet or exploiting the hICN forwarding pipeline (through the hICN plugin). In the following examples we show how to configure the packet generator in both cases. We use two forwarder A and B as in the previous example. However, both the client and server packet generator can run on the same vpp forwarder is needed.

IP Forwarding

Forwarder A (client)

```
sudo vppctl
vpp# set interface ip address TenGigabitEtherneta/0/0 2001::2/64
vpp# set interface state TenGigabitEtherneta/0/0 up
vpp# ip route add b001::/64 via 2001::3 TenGigabitEtherneta/0/0
vpp# ip route add 2001::3 via TenGigabitEtherneta/0/0
vpp# hicn pgen client src 2001::2 name b001::1/64 intfc TenGigabitEtherneta/0/0
vpp# exec /<path_to>pg.conf
vpp# packet-generator enable-stream hicn-pg
```

Where the file pg.conf contains the description of the stream to generate packets. In this case the stream sends 10 millions packets at a rate of 1Mpps

```
packet-generator new {
  name hicn-pg
  limit 10000000
  size 74-74
  node hicnpg-interest
  rate 1e6
  data {
    TCP: 5001::2 -> 5001::1
    hex 0x000000000000000050020000000001f4
    }
}
```

Forwarder B (server)

```
sudo vppctl
vpp# set interface ip address TenGigabitEtherneta/0/1 2001::3/64
vpp# set interface state TenGigabitEtherneta/0/1 up
vpp# hicn pgen server name b001::1/64 intfc TenGigabitEtherneta/0/1
```

hICN Forwarding

Forwarder A (client)

```
sudo vppctl
vpp# set interface ip address TenGigabitEtherneta/0/0 2001::2/64
vpp# set interface state TenGigabitEtherneta/0/0 up
vpp# ip route add b001::/64 via 2001::3 TenGigabitEtherneta/0/0
vpp# hicn enable b001::/64
vpp# create loopback interface
vpp# set interface state loop0 up
vpp# set interface ip address loop0 5002::1/64
vpp# ip neighbor loop0 5002::2 de:ad:00:00:00:00
vpp# hicn pgen client src 5001::2 name b001::1/64 intfc TenGigabitEtherneta/0/0
vpp# exec /<path_to>pg.conf
vpp# packet-generator enable-stream hicn-pg
```

The file pg.conf is the same showed in the previous example

Forwarder B (server)

```
sudo vppctl
vpp# set interface ip address TenGigabitEtherneta/0/1 2001::3/64
vpp# set interface state TenGigabitEtherneta/0/1 up
vpp# create loopback interface
vpp# set interface state loop0 up
vpp# set interface ip address loop0 2002::1/64
vpp# ip neighbor loop1 2002::2 de:ad:00:00:00:00
vpp# ip route add b001::/64 via 2002::2 loop0
vpp# hicn enable b001::/64
vpp# hicn pgen server name b001::1/64 intfc loop0
```

CHAPTER 8

Introduction

The transport library provides transport services and socket API for applications willing to communicate using the hICN protocol stack.

Overview:

- Implementation of the hICN core objects (interest, data, name..) exploiting the API provided by *libhicn*.
- IO modules for seamlessly connecting the application to the hicn-plugin for VPP or the *hicn-light* forwarder.
- Transport protocols (RAAQM, CBR, RTC)
- Transport services (authentication, integrity, segmentation, reassembly, naming)
- Interfaces for applications (from low-level interfaces for interest-data interaction to high level interfaces for Application Data Unit interaction)

CHAPTER 9

Build dependencies

9.1 Ubuntu

```
sudo apt install libasio-dev libconfig++-dev libssl-dev
```

If you wish to use the library for connecting to the vpp hicn-plugin, you will need to also install vpp and its libraries.

```
# Prevent vpp to set sysctl
export VPP_INSTALL_SKIP_SYSCTL=1
VPP_VERSION=$(cat "${VERSION_PATH}" | grep VPP_DEFAULT_VERSION | cut -d ' ' -f 2 | tr␣
↪-d '"' | grep -Po '\d\d.\d\d')

curl -s https://packagecloud.io/install/repositories/fdio/${VPP_VERSION//./}/script.
↪deb.sh | bash
curl -L https://packagecloud.io/fdio/${VPP_VERSION//./}/gpgkey | apt-key add -
sed -E -i 's/(deb.*)(\[.*\])(.*)/\1\3/g' /etc/apt/sources.list.d/fdio_${VPP_VERSION//.
↪/}.list
apt-get update

apt-get install -y \
  vpp-dev \
  libvppinfra-dev \
  vpp-plugin-core \
  vpp \
  libvppinfra
```

You can get them either from from the vpp packages or the source code. Check the VPP wiki for instructions.

9.2 macOS

We recommend to use HomeBrew for installing the libasio dependency:

```
brew install asio libconfig openssl@1.1
```

Since VPP does not support macOS, the IO module memif is not built.

CHAPTER 10

Build the library

The library is built by default from the main CMakeLists.txt. If you have all the dependencies installed, including *libhicn*, you can also build libtransport alone:

```
cd libtransport
mkdir build && cd build
cmake ..
cmake --build .
```

10.1 Compile options

The build process can be customized with the following options:

- `CMAKE_INSTALL_PREFIX`: The path where you want to install the library.
- `CMAKE_BUILD_TYPE`: The build configuration. Options: `Release`, `Debug`. Default is `Release`.
- `ASIO_HOME`: The folder containing the libasio headers.
- `VPP_HOME`: The folder containing the installation of VPP.

An option can be set using cmake -D`OPTION=VALUE`.

10.2 Install the library

For installing the library, from the cmake build folder:

```
cmake --build . -- install
```

CHAPTER 11

Usage

Examples on how to use the library can be found in the apps folder of the project. In particular you can check the **hiperf** application, which demonstrates how to use the API to interact with the hicn transport, both for consumer and producer.

11.1 Configuration file

The transport can be configured using a configuration file. There are two ways to tell libransport where to find the configuration file:

- programmatically - you set the configuration file path in your application:

```
// Set conf file path
std::string conf_file = "/opt/hicn/etc/transport.config"
// Parse config file
transport::interface::global_config::parseConfigurationFile(conf_file);
```

- using the environment variable TRANSPORT_CONFIG:

```
export TRANSPORT_CONFIG=/opt/hicn/etc/transport.config
./hiperf -C b001::1
```

Here is an example of configuration file:

```
// Configuration for io_module
io_module = {
  path = [];
  name = "forwarder_module";
};

// Configuration for forwarder io_module
forwarder = {
  n_threads = 1;
```

(continues on next page)

(continued from previous page)

```
  connectors = {
    c0 = {
      /* local_address and local_port are optional */
      local_address = "127.0.0.1";
      local_port = 33436;
      remote_address = "127.0.0.1";
      remote_port = 33436;
    }
  };

  listeners = {
    l0 = {
      local_address = "127.0.0.1";
      local_port = 33437;
    }
  };
};

// Logging
log = {
  // Log level (INFO (0), WARNING (1), ERROR (2), FATAL (3))
  minloglevel = 0;

  // Verbosity level for debug logs
  v= 2;

  // Log to stderr
  logtostderr = true;

  // Get fancy colored logs
  colorlogtostderr = true;

  // Log messages above this level also to stderr
  stderrthreshold = 2;

  // Set log prefix for each line log
  log_prefix = true;

  // Log dir
  log_dir = "/tmp";

  // Log only specific modules.
  // Example: "membuf=2,rtc=3"
  vmodule = "";

  // Max log size in MB
  max_log_size = 10;

  // Log rotation
  stop_logging_if_full_disk = true;
};
```

CHAPTER 12

Security

hICN has built-in authentication and integrity features by either:

- Cryptographically signing all packets using an asymmetric key (like RSA) or a symmetric one (like HMAC). The latter requires that all parties have prior access to the same key. Beware that this method is computationally expensive and impacts max throughput and CPU usage.
- Using manifests. Manifests are special packets that holds the digests of a group of data packets. Only the manifest needs to be signed and authenticated; other packets are authenticated simply by verifying that their digest is present in a manifest.

12.1 Per-packet signatures

To enable per-packet signature with asymmetric signing:

- On the producer, disable manifests (which are ON by default):

```
producer_socket->setSocketOption(GeneralTransportOptions::MANIFEST_MAX_CAPACITY,␣
↪0u);
```

- On the producer, instantiate an `AsymmetricSigner` object by passing either an asymmetric pair of keys as EVP_KEY object or a keystore path and password as strings:

```
std::shared_ptr<Signer> signer = std::make_shared<AsymmetricSigner>("./rsa.p12",
↪"hunter2");
```

- Pass the signer object to libtransport:

```
producer_socket->setSocketOption(GeneralTransportOptions::SIGNER, signer);
```

- On the consumer, instantiate an `AsymmetricVerifer` object by passing either a certificate as a X509 object, an asymmetric public key as a EVP_KEY object or a certificate path as a string:

```
std::shared_ptr<Verifier> verifier = std::make_shared<Verifier>("./rsa.crt");
```

- Pass the verifier object to libtransport:

```
consumer_socket->setSocketOption(GeneralTransportOptions::VERIFIER, verifier);
```

To enable per-packet signature with symmetric signing, follow the above steps replacing `AsymmetricSigner` with `SymmetricSigner` and `AsymmetricVerifer` with `SymmetricSigner`. A `SymmetricSigner` only has one constructor which expects a `CryptoSuite` and a passphrase. A `SymmetricVerifier` also has a single constructor which expects a passphrase:

```
std::shared_ptr<Signer> signer = std::make_shared<SymmetricSigner>(CryptoSuite::HMAC_
↪SHA256, "hunter2");
std::shared_ptr<Verifier> verifier = std::make_shared<SymmetricVerifier>("hunter2");
```

Check *Supported crypto suites* for the list of available suites.

12.2 Enabling manifests

- Follow steps 2-5 in *Per-packet signatures*.
- By default, a manifest has a maximum capacity `C_max` of 30 packets. To change this value:

```
producer_socket->setSocketOption(GeneralTransportOptions::MANIFEST_MAX_CAPACITY,␣
↪20u);
```

In the case of RTC, manifests are sent after the data they contain and on the consumer side, data packets are immediately forwarded to the application, *even if they weren't authenticated yet via a manifest*. This is to minimize latency. The digest of incoming data packets are kept in a buffer while waiting for manifests to arrive. When the buffer size goes above a threshold `T`, an alert is raised by the verifier object. That alert threshold is computed as follows:

```
T = manifest_factor_alert * C_max
```

The value of `C_max` is passed by the producer to the consumer at the start of the connection. `manifest_factor_alert` is a consumer socket option. It basically acts on the resilience of manifests against networks losses and reflects the application's tolerance to unverified packets: a higher value gives the transport the time needed to recover from several manifest losses but potentially allows a larger number of unverified packet to go the application before alerts are triggered. It is set to `20` by default and should always be `>= 1`. To change it:

```
consumer_socket_->setSocketOption(GeneralTransportOptions::MANIFEST_FACTOR_ALERT,␣
↪10u);
```

The buffer does not keep unverified packets indefinitely. After a certain amount of packets have been received and processed (and were verified or not), older packets still unverified are flushed out. This is to prevent the buffer to grow uncontrollably and to raise alerts for packets that are not relevant to the application anymore. That threshold of relevance is computed as follows:

```
T = manifest_factor_relevant * C_max
```

`manifest_factor_relevant` is a consumer socket option. It is set to `100` by default. Its value must be set so that `manifest_factor_relevant > manifest_factor_alert >= 1`. If `manifest_factor_relevant <= manifest_factor_alert`, no alert will ever be raised. To change it:

```
consumer_socket_->setSocketOption(GeneralTransportOptions::MANIFEST_FACTOR_RELEVANT,␣
↪200u);
```

12.3 Handling authentication failures

When a data packet fails authentication, or when the unverified buffer is full in the case of RTC, an alert is triggered by the verifier object. By default libtransport aborts the connection upon reception of that alert. You may want to intercept authentication failures in your application:

- Define a callback with arguments an `uint32_t` integer, which will be set to the suffix of the faulty packet, and a `auth::VerificationPolicy`, which will be set to the action suggested by the verifier object. The callback must return another `auth::VerificationPolicy` which will be the actual action taken by libtransport:

```
auth::VerificationPolicy onAuthFailed(uint32_t suffix, auth::VerificationPolicy␣
↪policy) {
  std::cout << "auth failed for packet " << suffix << std::endl;
  return auth::VerificationPolicy::ACCEPT;
}
```

- Give that callback to your `Verifier` object as well as a list of `auth::VerificationPolicy` to intercept (if left empty, will be set by default to `{ABORT, DROP}`):

```
verifier->setVerificationFailedCallback(&onAuthFailed, {
  auth::VerificationPolicy::ABORT,
  auth::VerificationPolicy::DROP,
  auth::VerificationPolicy::UNKNOWN,
});
```

12.4 Supported crypto suites

The following `CryptoSuite` are supported by libtransport:

```
ECDSA_BLAKE2B512
ECDSA_BLAKE2S256
ECDSA_SHA256
ECDSA_SHA512
RSA_BLAKE2B512
RSA_BLAKE2S256
RSA_SHA256
RSA_SHA512
HMAC_BLAKE2B512
HMAC_BLAKE2S256
HMAC_SHA256
HMAC_SHA512
DSA_BLAKE2B512
DSA_BLAKE2S256
DSA_SHA256
DSA_SHA512
```

CHAPTER 13

Logging

Internally libtransport uses glog as logging library. If you want to have a more verbose transport log when launching a test or an app, you can set environment variables in this way:

```
GLOG_v=4 hiperf -S b001::/64
```

For a more exhaustive list of options, please check the instructions in the glog README.

Useful options include enabling logging *per module*. Also you can compile out useless messages in release builds.

CHAPTER 14

Portable forwarder

14.1 Introduction

hicn-light is a portable forwarder that makes use of IPC and standard sockets to communicate.

14.2 Using hicn-light

14.2.1 Dependencies

Build dependencies:

- C11 (clang / gcc)
- CMake 3.10

Basic dependencies:

- OpenSSL
- pthreads
- libevent

14.3 hicn-light executables

hicn-light is a set of binary executables that are used to run a forwarder instance. The forwarder can be run and configured using the commands:

- `hicn-light-daemon`
- `hicn-light-control`

Use the `-h` option to display the help messages.

14.3.1 hicn-light daemon

The command `hicn-light-daemon` runs the hicn-light forwarder. The forwarder can be executed with the following options:

```
hicn-light-daemon [--port port] [--daemon] [--capacity objectStoreSize] [--log level]
                [--log-file filename] [--config file]

Options:
--port <tcp_port>               = tcp port for local in-bound connections
--daemon                        = start as daemon process
--capacity <objectStoreSize>    = maximum number of content objects to cache. To␣
↪disable the cache
                                  objectStoreSize must be 0.
                                  Default vaule for objectStoreSize is 100000
--log <log_granularity>         = sets the log level. Available levels: trace, debug,␣
↪info, warn, error, fatal
--log-file <output_logfile>     = file to write log messages to (required in daemon␣
↪mode)
--config <config_path>          = configuration filename
```

The configuration file contains configuration lines as per hicn-light-control (see below for all the available commands). If logging level or content store capacity is set in the configuration file, it overrides the command_line.

In addition to the listeners setup in the configuration file, hicn-light-daemon will listen on TCP and UDP ports specified by the –port flag (or default port). It will listen on both IPv4 and IPv6 if available. The default port for hicn-light is 9695.

14.3.2 hicn-light-control

`hicn-light-control` can be used to send command to the hicn-light forwarder and configure it. The command can be executed in the following way:

```
hicn-light-control [commands]

Options:
    -h                          = This help screen
    commands                    = configuration line to send to hicn-light (use 'help' for␣
↪list)
```

Available commands in hicn-light-control

This is the full list of available commands in `hicn-light-control`. This commands can be used from the command line running `hicn-light-control` as explained before, or listing them in a configuration file.

The list of commands can be navigated using `hicn-light-control help`, `hicn-light-control help <object>`, `hicn-light-control help <object> <action>`.

`add listener`: creates a TCP or UDP listener with the specified options on the local forwarder. For local connections (application to hicn-light) we expect a TCP listener. The default port for the local listener is 9695.

```
add listener <protocol> <symbolic> <local_address> <local_port> <interface>

  <symbolic>          :User defined name for listener, must start with alpha and␣
↪bealphanum
```

(continues on next page)

(continued from previous page)

```
  <protocol>          :tcp | udp
  <localAddress>      :IPv4 or IPv6 address
  <local_port>        :TCP/UDP port
  <interface>         :interface on which to bind
```

`add connection`: creates a TCP or UDP connection on the local forwarder with the specified options.

```
add connection <protocol> <symbolic> <remote_ip> <remote_port> <local_ip> <local_port>

  <protocol>              : tcp | udp
  <symbolic>              : symbolic name, e.g. 'conn1' (must be unique, start with␣
↪alpha)
  <remote_ip>             : the IPv4 or IPv6 of the remote system
  <remote_port>           : the remote TCP/UDP port
  <local_ip>              : local IP address to bind to
  <local_port>            : local TCP/UDP port
```

`list`: lists the connections, routes or listeners available on the local hicn-light forwarder.

```
list <connections | routes | listeners>
```

`add route`: adds a route to the specified connection.

```
add route <symbolic | connid> <prefix> <cost>

  <symbolic>    :The symbolic name for an exgress (must be unique, start with alpha)
  <connid>:     :The egress connection id (see 'list connection' command)
  <prefix>:     :ipAddress/netmask
  <cost>:       :positive integer representing cost
```

`remove connection`: removes the specified connection. At the moment, this commands is available only for UDP connections, TCP is ignored.

```
remove connection <symbolic | connid>

  <symbolic>    :The symbolic name for an exgress (must be unique, start with alpha)
  <connid>:     :The egress connection id (see 'list connection' command)
```

`remove route`: remove the specified prefix for a local connection.

```
remove route <symbolic | connid> <prefix>

  <connid>     : the alphanumeric name of a local connection
  <prefix>     : the prefix (ipAddress/netmask) to remove
```

`serve cache`: enables/disables replies from local content store (if available).

```
serve cache <on|off>
```

`store cache`: enables/disables the storage of incoming data packets in the local content store (if available).

```
store cache <on|off>
```

`clear cache`: removes all the cached data form the local content store (if available).

```
clear cache
```

`set strategy`: sets the forwarding strategy for a give prefix. There are 4 different strategies implemented in hicn-light:

- **random**: each interest is forwarded randomly to one of the available output connections.
- **loadbalancer**: each interest is forwarded toward the output connection with the lowest number of pending interests. The pending interest are the interest sent on a certain connection but not yet satisfied. More information are available in: G. Carofiglio, M. Gallo, L. Muscariello, M. Papalini, S. Wang, "Optimal multipath congestion control and request forwarding in information-centric networks", ICNP 2013.
- **low_latency**: uses the face with the lowest latency. In case more faces have similar latency the strategy uses them in parallel.
- **replication**
- **bastpath**

```
set strategy <prefix> <strategy>

  <preifx>    : the prefix to which apply the forwarding strategy
  <strategy>  : random | loadbalancer | low_latency | replication | bestpath
```

`set wldr`: turns on/off WLDR on the specified connection. WLDR (Wireless Loss Detiection and Recovery) is a protocol that can be used to recover losses generated by unreliable wireless connections, such as WIFI. More information on WLDR are available in: G. Carofiglio, L. Muscariello, M. Papalini, N. Rozhnova, X. Zeng, "Leveraging ICN In-network Control for Loss Detection and Recovery in Wireless Mobile networks", ICN 2016. Notice that WLDR is currently available only for UDP connections. In order to work properly, WLDR needs to be activated on both side of the connection.

```
set wldr <on|off> <symbolic | connid>

  <symbolic>   :The symbolic name for an exgress (must be unique, start with alpha)
  <connid>:    :The egress connection id (see 'help list connections')
```

`set mapme`: enables/disables mapme, enables/disables mapme discovery, set the timescale value (expressed in milliseconds), set the retransmission time value (expressed in milliseconds).

```
mapme set enable <on|off>
mapme set discovery <on|off>
mapme set timescale <milliseconds>
mapme set retx <milliseconds>
```

`quit`: exits the interactive bash.

14.3.3 hicn-light configuration file example

This is an example of a simple configuration file for hicn-light. It can be loaded by running the command `hicn-light-daemon --config configFile.cfg`, assuming the file name is `configFile.cfg`.

```
# Create a local listener on port 9199. This will be used by the applications to talk␣
↪with the forwarder
add listener udp local0 192.168.0.1 9199 eth0

# Create a connection with a remote hicn-light-daemon, with a listener on 192.168.0.
↪20 12345
add connection udp conn0 192.168.0.20 12345 192.168.0.1 9199 eth0

```

(continues on next page)

(continued from previous page)

```
# Add a route toward the remote node
add route conn0 c001::/64 1
```

CHAPTER 15

Face manager

15.1 Overview

The architecture of the face manager is built around the concept of interfaces, which allows for a modular and extensible deployment.

Interfaces are used to implement in isolation various sources of information which help with the construction of faces (such as network interface and service discovery), and with handling the heterogeneity of host platforms.

15.1.1 Platform and supported interfaces

Currently, Android, Linux and MacOS are supported through the following interfaces:

- hicn-light [Linux, Android, MacOS, iOS] An interface to the hicn-light forwarder, and more specifically to the Face Table and FIB data structures. This component is responsible to effectively create, update and delete faces in the forwarder, based on the information provided by third party interfaces, plus adding default routes for each of the newly created face. The communication with the forwarder is based on the hicn control library (`libhicnctrl`).
- netlink [Linux, Android] The default interface on Linux systems (including Android) to communicate with the kernel and receive information from various sources, including link and address information (both IPv4 and IPv6) about network interfaces.
- android_utility [Android only] Information available through Netlink is limited with respect to cellular interfaces. This component allows querying the Android layer through SDK functions to get the type of a given network interface (Wired, WiFi or Cellular).
- bonjour [Linux, Android] This component performs remote service discovery based on the bonjour protocol to discover a remote hICN forwarder that might be needed to establish overlay faces.
- network_framework [MacOS, iOS]

  This component uses the recommended Network framework on Apple devices, which provided all required information to query faces in a unified API: link and address information, interface types, and bonjour service discovery.

15.2 Developing a new interface

15.2.1 Dummy template

The face manager source code includes a template that can be used as a skeleton to develop new faces. It can be found in `src/interface/dummy/dummy.{h,c}`. Both include guard and specific interface functions are prefixed by a (short) identifier which acts as a namespace for interface specific code (in our case the string 'dummy_').

Registration and instantiation of the different interfaces is currently done at compile time in the file `src/api.c`, and the appropriate hooks to use the dummy interface are available in the code between `#if 0/#endif` tags.

Interface template header and configuration parameters

All interfaces have a standard interface defined in `src/interface.{h,c}`, and as such the header file is only used to specify the configuration parameters of the interface, if any.

In the template, these configuration options are empty:

```
/*
 * Configuration data
 */
typedef struct {
    /* ... */
} dummy_cfg_t;
```

Overview of the interface template

The file starts with useful includes:

```
- the global include `<hicn/facemgr.h>` : this provides public facing elements
    of the face manager, such the standard definition of faces (`face_t` from
    `libhicnctrl`), helper classes (such as `ip_address_t` from `libhicn`), etc.
- common.h
- facelet.h : facelets are the basic unit of communication between the face
manager and the different interfaces. They are used to construct the faces
incrementally.
- interface.h : the parent class of interfaces, such as the current dummy
interface.
```

Each interface can hold a pointer to an internal data structure, which is declared as follows:

```
/*
 * Internal data
 */
typedef struct {
    /* The configuration data will likely be allocated on the stack (or should
     * be freed) by the caller, we recommend to make a copy of this data.
     * This copy can further be altered with default values.
     */
    dummy_cfg_t cfg;

    /* ... */

    int fd; /* Sample internal data: file descriptor */
} dummy_data_t;
```

We find here a copy of the configuration settings (which allows the called to instantiate the structure on the stack), as well as a file descriptor (assuming most interfaces will react on events on a file descriptor).

The rest of the file consists in the implementation of the interface, in particular the different function required by the registration of a new interface to the system. They are grouped as part of the `interface_ops_t` data structure declared at the end of the file:

```
interface_ops_t dummy_ops = {
    .type = "dummy",
    .initialize = dummy_initialize,
    .finalize = dummy_finalize,
    .callback = dummy_callback,
    .on_event = dummy_on_event,
};
```

The structure itself is declared and documented in `src/interface.h`

```
/**
 * \brief Interface operations
 */
typedef struct {
    /** The type given to the interfaces */
    char * type;
    /* Constructor */
    int (*initialize)(struct interface_s * interface, void * cfg);
    /* Destructor */
    int (*finalize)(struct interface_s * interface);
    /* Callback upon file descriptor event (iif previously registered) */
    int (*callback)(struct interface_s * interface);
    /* Callback upon facelet events coming from the face manager */
    int (*on_event)(struct interface_s * interface, const struct facelet_s * facelet);
} interface_ops_t;
```

Such an interface has to be registered first, then one (or multiple) instance(s) can be created (see `src/interface.c` for the function prototypes, and `src/api.c` for their usage).

- interface registration:

```
extern interface\_ops\_t dummy\_ops;

/* [...] */

rc = interface\_register(&dummy\_ops);
if (rc < 0)
    goto ERR_REGISTER;
```

- interface instantiation:

```
#include "interfaces/dummy/dummy.h"

/* [...] */

rc = facemgr_create_interface(facemgr, "dummy0", "dummy", &facemgr->dummy);
if (rc < 0) {
    ERROR("Error creating 'Dummy' interface\n");
    goto ERR_DUMMY_CREATE;
}
```

Implementation of the interface API

We now quickly go other the different functions, but their usage will be better understood through the hands-on example treated in the following section.

In the template, the constructor is the most involved as it need to:

- initialize the internal data structure:

```
    dummy_data_t * data = malloc(sizeof(dummy_data_t));
    if (!data)
        goto ERR_MALLOC;
    interface->data = data;
```

- process configuration parameters, eventually setting some default values:

```
    /* Use default values for unspecified configuration parameters */
    if (cfg) {
        data->cfg = *(dummy_cfg_t *)cfg;
    } else {
        memset(&data->cfg, 0, sizeof(data->cfg));
    }
```

- open an eventually required file descriptor

For the sake of simplicity, the current API only supports a single file descriptor per-interface, and it has to be created in the constructor, and set as the return value so as to be registered by the system, and added to the event loop for read events. A return value of 0 means the interface does not require any file descriptor. As usual, a negative return value indicates an error.

```
    data->fd = 0;

    /* ... */

    /*
     * We should return a negative value in case of error, and a positive value
     * otherwise:
     *  - a file descriptor (>0) will be added to the event loop; or
     *  - 0 if we don't use any file descriptor
     */
    return data->fd;
```

While support for multiple file descriptors might be added in the future, an alternative short-term implementation might consider the instantiation of multiple interface, as is done for Bonjour in the current codebase, in `src/api.c`.

Data reception on the file descriptor will get the callback function called, in our case `dummy_callback`. Finally, the destructor `dummy_finalize` should close an eventual open file descriptor.

In order to retrieve the internal data structure, that should in particular store such a file descriptor, all other function but the constructor can dereference it from the interface pointer they receive as parameter:

```
dummy_data_t * data = (dummy_data_t*)interface->data;
```

Raising and receiving events

An interface will receive events in the form of a facelet through the `*_on_event` function. It can then use the facelet API we have described above to read information about the face.

As this information is declared const, the interface can either create a new facelet (identified by the same netdevice and protocol family), or eventually clone it.

The facelet event can then be defined and raised to the face manager for further processing through the following code:

```
    facelet_set_event(facelet, EVENT_TYPE_CREATE);
    interface_raise_event(interface, facelet);
```

Here the event is a facelet creation (`EVENT_TYPE_CREATE`). The full facelet API and the list of possible event types is available in `src/facelet.h`

Integration in the build system

The build system is based on CMake. Each interface should declare its source files, private and public header files, as well as link dependencies in the local `CMakeLists.txt` file.

15.2.2 Hands-on

Architecture

In order to better illustrate the development of a new interface, we will consider the integration of a sample server providing a signal instructing the face manager to alternatively use either the WiFi or the LTE interface. The code of this server is available in the folder `examples/updownsrv/`, and the corresponding client code in `examples/updowncli`.

Communication between client and server is done through unix sockets over an abstract namespace (thereby not using the file system, which would cause issues on Android). The server listens for client connections, and periodically broadcast a binary information to all connected clients, in the form of one byte equal to either \0 (which we might interpret as enable LTE, disable WiFi), or \1 (enable WiFi, disable LTE).

Our objective is to develop a new face manager interface that would listen to such event in order to update the administrative status of the current faces. This would thus alternatively set the different interfaces administratively up and down (which takes precedence over the actual status of the interface when the forwarder establishes the set of available next hops for a given prefix). The actual realization of such queries will be ultimately performed by the hicn-light interface.

Sample server and client

In the folder containing the source code of hICN, the following commands allow to run the sample server:

```
cd ctrl/facemgr/examples/updownsrv
make
./updownsrv
```

The server should display "Waiting for clients…"

Similar commands allow to run the sample client:

```
cd ctrl/facemgr/examples/updowncli
make
./updowncli
```

The client should display "Waiting for server data…", then every couple of seconds display either "WiFi" or "LTE".

Face manager interface

An example illustrating how to connect to the dummy service from `updownsrv` is provided as the `updown` interface in the facemgr source code.

This interface periodically swaps the status of the LTE interface up and down. It is instantiated as part of the facemgr codebase when the code is compiled with the ‘‘-DWITH_EXAMPLE_UPDOWN‘ cmake option.

CHAPTER 16

Control plane support

A new control plane for hicn is under construction.

CHAPTER 17

Telemetry

Tools to collect telemetry from hICN forwarders.

17.1 Introduction

The project contains two plugins for collectd:

- vpp: to collect statistics for VPP
- vpp-hicn: to collect statistics for hICN

Currently the two plugins provide the following functionalities:

- vpp: statistics (rx/tx bytes and packets) for each available interface.
- vpp-hicn: statistics (rx/tx bytes and packets) for each available face.

17.2 Quick start

From the code tree root:

```
cd telemetry
mkdir -p build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX=/usr
make
sudo make install
```

17.3 Using hICN collectd plugins

17.3.1 Platforms

hICN collectd plugins have been tested in:

- Ubuntu 20.04 LTS

17.3.2 Dependencies

Build dependencies:

VPP 22.02, Debian packages can be found on packagecloud:

- vpp
- libvppinfra-dev
- vpp-dev
- hicn-plugin-dev
- `collectd` and `collectd-dev`: `sudo apt install collectd collectd-dev libyajl-dev`

17.4 Getting started

Collectd needs to be configured in order to use the hICN plugins. To enable the plugins, add the following lines to `/etc/collectd/collectd.conf`:

```
LoadPlugin vpp
LoadPlugin vpp_hicn
```

Before running collectd, a vpp forwarder must be started. If the vpp-hicn plugin is used, the hicn-plugin must be available in the vpp forwarder.

If you need the custom types that the two plugins define, they are present in `telemetry/custom_types.db`. It is useful if you are using InfluxDB as it requires the type database for multi-value metrics (see CollectD protocol support in InfluxDB).

17.5 Plugin options

`vpp` and `vpp-hicn` have the same two options:

- `Verbose` enables additional statistics. You can check the sources to have an exact list of available metrics.
- `Tag` tags the data with the given string. Useful for identifying the context in which the data was retrieved in InfluxDB for instance. If the tag value is `None`, no tag is applied.

17.5.1 Example: storing statistics from vpp and vpp-hicn

We'll use the rrdtool and csv plugins to store statistics from vpp and vpp-hicn. Copy the configuration below in a file called `collectd.conf` and move it to `/etc/collectd`:

```
######################################################################
# Global                                                             #
######################################################################
FQDNLookup true
BaseDir "/var/lib/collectd"
Interval 1
# if you are using custom_types.db, you can specify it
TypesDB "/usr/share/collectd/types.db" "/etc/collectd/custom_types.db"

######################################################################
# Logging                                                            #
######################################################################
LoadPlugin logfile

<Plugin logfile>
  LogLevel "info"
  File "/var/log/collectd.log"
  Timestamp true
  PrintSeverity true
</Plugin>

######################################################################
# Plugins                                                            #
######################################################################
LoadPlugin csv
LoadPlugin rrdtool
LoadPlugin vpp
LoadPlugin vpp_hicn

######################################################################
# Plugin configuration                                               #
######################################################################
<Plugin csv>
  DataDir "/var/lib/collectd/csv"  # the folder where statistics are stored in csv
  StoreRates true
</Plugin>

<Plugin rrdtool>
  DataDir "/var/lib/collectd/rrd"  # the folder where statistics are stored in rrd
</Plugin>

<Plugin vpp>
  Verbose true
  Tag "None"
</Plugin>

<Plugin vpp_hicn>
  Verbose true
  Tag "None"
</Plugin>
```

Run vpp and collectd:

```
systemctl start vpp
systemctl start collectd
```

CHAPTER 18

Utility applications

18.1 Introduction

hicn-ping-server, hicn-ping-client and hiperf are three utility applications for testing and benchmarking stack.

18.2 Using hICN utils applications

18.2.1 Dependencies

Build dependencies:

- C++14 (clang++ / g++)
- CMake 3.4

Basic dependencies:

- OpenSSL
- pthreads
- libevent
- libhicntransport

18.3 Executables

The utility applications are a set of binary executables consisting of a client/server ping applications (hicn-ping-server and hicn-ping-client) and a hicn implementation of iPerf (hiperf).

18.3.1 hicn-ping-server

The command `hicn-ping-server` runs the server side ping application. `hicn-ping-server` can be executed with the following options:

```
usage: hicn-ping-server [options]

Options:
-s <content_size>            = object content size (default 1350B)
-n <hicn_name>               = hicn name (default b001::/64)
-f                           = set tcp flags according to the flag received (default␣
↪false)
-l <lifetime>                = data lifetime
-r                           = always reply with a reset flag (default false)
-t <ttl>                     = set ttl (default 64)
-d                           = daemon mode
-H                           = help
```

Example:

```
hicn-ping-server -n c001::/64
```

18.3.2 hicn-ping-client

The command `hicn-ping-client` runs the client side ping application. `hicn-ping-client` can be executed with the following options:

```
usage: hicn-ping-client [options]

Options:
-i <ping_interval>           = ping interval in microseconds (default 1000000ms)
-m <max_pings>               = maximum number of pings to send (default 10)
-s <source_port>             = source port (default 9695)
-d <destination_port>        = destination port (default 8080)
-t <ttl>                     = set packet ttl (default 64)
-O                           = open tcp connection (three way handshake) (default␣
↪false)
-S                           = send always syn messages (default false)
-A                           = send always ack messages (default false)
-n <hicn_name>               = hicn name (default b001::1)
-l <lifetime>                = interest lifetime in milliseconds (default 500ms)
-H                           = help
```

Example:

```
hicn-ping-client -n c001::1
```

18.3.3 hiperf

The command `hiperf` is a tool for performing network throughput measurements with hicn. It can be executed as server or client using the following options:

```
HIPERF - Instrumentation tool for performing active networkmeasurements with hICN
usage: hiperf [-S|-C] [options] [prefix|name]
```

(continues on next page)

(continued from previous page)

```
SERVER OR CLIENT:
-D                                        Run as a daemon
-R                                        Run RTC protocol (client or server)
-f      <filename>                        Log file
-z      <io_module>                       IO module to use. Default: hicnlightng_module
-F      <conf_file>                       Path to optional configuration file for␣
↪libtransport
-a                                        Enables data packet aggregation. Works only␣
↪in RTC mode
-X      <param>                           Set FEC params. Options are Rely_K#_N# or RS_K
↪#_N#

SERVER SPECIFIC:
-A      <content_size>                    Sends an application data unit in bytes that␣
↪is published once before exit
-s      <packet_size>                     Data packet payload size.
-r                                        Produce real content of <content_size> bytes
-m      <manifest_capacity>               Produce transport manifest
-l                                        Start producing content upon the reception of␣
↪the first interest
-K      <keystore_path>                   Path of p12 file containing the crypto␣
↪material used for signing packets
-k      <passphrase>                      String from which a 128-bit symmetric key␣
↪will be derived for signing packets
-p      <password>                        Password for p12 keystore
-y      <hash_algorithm>                  Use the selected hash algorithm for computing␣
↪manifest digests (default: SHA256)
-x                                        Produces application data units of size
↪<content_size> without resetting the name suffix to 0.
-B      <bitrate>                         RTC producer data bitrate, to be used with␣
↪the -R option.
-I                                        Interactive mode, start/stop real time␣
↪content production by pressing return. To be used with the -R option
-T      <filename>                        Trace based mode, hiperf takes as input a␣
↪file with a trace. Each line of the file indicates the timestamp and the size of␣
↪the packet to generate. To be used with the -R option. -B and -I will be ignored.
-E                                        Enable encrypted communication. Requires the␣
↪path to a p12 file containing the crypto material used for the TLS handshake
-G      <port>                            Input stream from localhost at the specified␣
↪port

CLIENT SPECIFIC:
-b      <beta_parameter>                  RAAQM beta parameter
-d      <drop_factor_parameter>           RAAQM drop factor parameter
-L      <interest lifetime>               Set interest lifetime.
-u      <delay>                           Set max lifetime of unverified packets.
-M      <input_buffer_size>               Size of consumer input buffer. If 0,␣
↪reassembly of packets will be disabled.
-W      <window_size>                     Use a fixed congestion window for retrieving␣
↪the data.
-i      <stats_interval>                  Show the statistics every <stats_interval>␣
↪milliseconds.
-c      <certificate_path>                Path of the producer certificate to be used␣
↪for verifying the origin of the packets received.
-k      <passphrase>                      String from which is derived the symmetric␣
↪key used by the producer to sign packets and by the consumer to verify them.
```

(continues on next page)

(continued from previous page)

```
-t                                        Test mode, check if the client is receiving␣
↪the correct data. This is an RTC specific option, to be used with the -R (default:␣
↪false)
-P                                        Prefix of the producer where to do the␣
↪handshake
-j      <relay_name>                      Publish received content under the name relay_
↪name.This is an RTC specific option, to be used with the -R (default: false)
-g      <port>                            Output stream to localhost at the specified␣
↪port
-e      <strategy>                        Enance the network with a realiability␣
↪strategy. Options 1: unreliable, 2: rtx only, 3: fec only, 4: delay based, 5: low␣
↪rate, 6: low rate and best path 7: low rate and replication, 8: low rate and best␣
↪path/replication(default: 2 = rtx only)
-H                                        Disable periodic print headers in stats␣
↪report.
-n      <nb_iterations>                   Print the stats report <nb_iterations> times␣
↪and exit.
                                          This option limits the duration of the run to
↪<nb_iterations> * <stats_interval> milliseconds.
```

Example:

```
hiperf -S b001::/64
hiperf -C b001::
```

18.4 Client/Server benchmarking using `hiperf`

18.4.1 hicn-light-daemon

This tutorial will explain how to configure a simple client-server topology and retrieve network measurements using the hiperf utility.

We consider this simple topology, consisting on two linux VM which are able to communicate through an IP network (you can also use containers or physical machines):

```
|client (10.0.0.1/24; 9001::1/64)|======|server (10.0.0.2/24; 9001::2/64)|
```

Install the hICN suite on two linux VM. This tutorial makes use of Ubuntu 18.04, but it could easily be adapted to other platforms. You can either install the hICN stack using binaries or compile the code. In this tutorial we will build the code from source.

```
apt-get update && apt-get install -y curl
curl -s https://packagecloud.io/install/repositories/fdio/release/script.deb.sh |␣
↪sudo bash
apt-get install -y git \
                   cmake \
                   build-essential \
                   libasio-dev \
                   libcurl4-openssl-dev \
                   --no-install-recommends
mkdir hicn-suite && cd hicn-suite
git clone https://github.com/FDio/hicn.git hicn-src
mkdir hicn-build && cd hicn-build
```

(continues on next page)

(continued from previous page)

```
cmake ../hicn-src -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=../hicn-install -
↪DBUILD_APPS=ON
make -j4 install
export HICN_ROOT=${PWD}/../hicn-install
```

It should install the hICN suite under hicn-install.

hicn-light forwarder with UDP faces

Server configuration

Create a configuration file for the hicn-light forwarder. Here we are configuring UDP faces.

```
server$ mkdir -p ${HICN_ROOT}/etc
server$ LOCAL_IP="10.0.0.1" # Put here the actual IPv4 of the local interface
server$ LOCAL_PORT="12345"
server$ cat << EOF > ${HICN_ROOT}/etc/hicn-light.conf
add listener udp list0 ${LOCAL_IP} ${LOCAL_PORT}
EOF
```

Start the hicn-light forwarder:

```
server$ sudo ${HICN_ROOT}/bin/hicn-light-daemon --daemon --capacity 0 --log-file $
↪{HICN_ROOT}/hicn-light.log --config ${HICN_ROOT}/etc/hicn-light.conf
```

We set the forwarder capacity to 0 because we want to measure the end-to-end performance without retrieving any data packet from intermediate caches.

Run the *hiperf* server:

```
server$ ${HICN_ROOT}/bin/hiperf -S b001::/64
```

The hiperf server will register the prefix b001::/64 on the local forwarder and will reply with pre-allocated data packet. In this test we won't consider segmentation and reassembly cost.

Client configuration

Create a configuration file for the hicn-light forwarder at the client. Here we are configuring UDP faces.

```
client$ mkdir -p ${HICN_ROOT}/etc
client$ LOCAL_IP="10.0.0.2" # Put here the actual IPv4 of the local interface
client$ LOCAL_PORT="12345"
client$ REMOTE_IP="10.0.0.1" # Put here the actual IPv4 of the remote interface
client$ REMOTE_PORT="12345"
client$ cat << EOF > ${HICN_ROOT}/etc/hicn-light.conf
add listener udp list0 ${LOCAL_IP} ${LOCAL_PORT}
add connection udp conn0 ${REMOTE_IP} ${REMOTE_PORT} ${LOCAL_IP} ${LOCAL_PORT}
add route conn0 b001::/16 1
EOF
```

Run the hicn-light forwarder:

```
client$ sudo ${HICN_ROOT}/bin/hicn-light-daemon --daemon --capacity 1000 --log-file $
↪{HICN_ROOT}/hicn-light.log --config ${HICN_ROOT}/etc/hicn-light.conf
```

Run the *hiperf* client:

```
client$ ${HICN_ROOT}/bin/hiperf -C b001::1 -W 50
EOF
```

This will run the client with a fixed window of 50 interests.

hicn-light forwarder with hICN faces

For sending hICN packets directly over the network, using hicn faces, change the configuration of the two forwarders and restart them.

Server

```
server$ mkdir -p ${HICN_ROOT}/etc
server$ LOCAL_IP="9001::1"
server$ cat << EOF > ${HICN_ROOT}/etc/hicn-light.conf
add listener hicn lst 0::0
add punting lst b001::/16
add listener hicn list0 ${LOCAL_IP}
EOF
```

Client

```
client$ mkdir -p ${HICN_ROOT}/etc
client$ LOCAL_IP="9001::2"
client$ REMOTE_IP="9001::1"
client$ cat << EOF > ${HICN_ROOT}/etc/hicn-light.conf
add listener hicn lst 0::0
add punting lst b001::/16
add listener hicn list0 ${LOCAL_IP}
add connection hicn conn0 ${REMOTE_IP} ${LOCAL_IP}
add route conn0 b001::/16 1
EOF
```

18.4.2 VPP based hicn-plugin

In this example we will do a local hiperf client-server communication. First, we need to compile the hicn stack and enable VPP support:

```
apt-update && apt-get install -y curl
curl -s https://packagecloud.io/install/repositories/fdio/release/script.deb.sh |␣
↪sudo bash
apt-get install -y git \
                   cmake \
                   build-essential \
                   libasio-dev \
                   vpp vpp-dev vpp-plugin-core libvppinfra \
                   libmemif libmemif-dev \
                   python3-ply \
                   --no-install-recommends
mkdir hicn-suite && cd hicn-suite
```

(continues on next page)

(continued from previous page)

```
git clone https://github.com/FDio/hicn.git hicn-src
mkdir hicn-build && cd hicn-build
cmake ../hicn-src -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/usr -DBUILD_
↪APPS=ON -DBUILD_HICNPLUGIN=ON
sudo make -j 4 install
export HICN_ROOT=${PWD}/../hicn-install
```

Make sure vpp is running:

```
sudo systemctl restart vpp
```

Run the hicn-plugin:

```
vppctl hicn control start
```

Run hiperf server:

```
hiperf -S b001::/64
```

Run hiperf client:

```
hiperf -C b001::1 -W 300
```

CHAPTER 19

Applications

The open source distribution provides a few application examples: a MPEG-DASH video player, a HTTP reverse proxy, a command line HTTP GET client.

hICN sockets have been successfully used to serve HTTP, RTP and RSockets application protocols.

19.1 Dependencies

Build dependencies:

- C++14 (clang++ / g++)
- CMake 3.5 or higher

Basic dependencies:

- OpenSSL
- pthreads
- libevent
- libcurl
- libhicntransport

19.2 Executables

19.2.1 hicn-http-proxy

`hicn-http-proxy` is a reverse proxy which can be used for augmenting the performance of a legacy HTTP/TCP server by making use of hICN. It performs the following operations:

- Receive a HTTP request from a hICN client

- Forward it to a HTTP server over TCP
- Receive the response from the server and send it back to the client

```
hicn-http-proxy [HTTP_PREFIX] [OPTIONS]

HTTP_PREFIX: The prefix used for building the hicn names.

Options:
-a <server_address>   = origin server address
-p <server_port>      = origin server port
-c <cache_size>       = cache size of the proxy, in number of hicn data packets
-m <mtu>              = mtu of hicn packets
-P <prefix>           = optional most significant 16 bits of hicn prefix, in␣
↪hexadecimal format
```

Example:

```
./hicn-http-proxy http://webserver -a 127.0.0.1 -p 8080 -c 10000 -m 1200 -P b001
```

The hICN names used by the hicn-http-proxy for naming the HTTP responses are composed in the following way, starting from the most significant byte:

- The first 2 bytes are the prefix specified in the -P option
- The next 6 bytes are the hash (Fowler–Noll–Vo non-crypto hash) of the locator (in the example `webserver`, without the `http://` part)
- The last 8 bytes are the hash (Fowler–Noll–Vo non-crypto hash) of the http request corresponding to the response being forwarded back to the client.

19.2.2 higet

Higet is a non-interactive HTTP client working on top oh hICN.

```
higet [option]... [url]
Options:
-O <output_path>            = write documents to <output_file>. Use '-' for stdout.
-S                          = print server response.
-P                          = optional first 16 bits of hicn prefix, in hexadecimal␣
↪format

Example:
./higet -P b001 -O - http://webserver/index.html
```

The hICN names used by higet for naming the HTTP requests are composed the way described in *hicn-http-proxy*.

19.3 HTTP client-server with hicn-http-proxy

We consider the following topology, consisting on two linux VMs which are able to communicate through an IP network (you can also use containers or physical machines):

```
|client (10.0.0.1/24; 9001::1/64)|======|server (10.0.0.2/24; 9001::2/64)|
```

Install the hICN suite on two linux VM. This tutorial makes use of Ubuntu 18.04, but it could easily be adapted to other platforms. You can either install the hICN stack using binaries or compile the code. In this tutorial we will make use of docker container and binaries packages.

The client will use of the hicn-light forwarder, which is lightweight and tailored for devices such as android and laptops. The server will use the hicn-plugin of vpp, which guarantees better performances and it is the best choice for server applications.

Keep in mind that on the same system the stack based on vpp forwarder cannot coexist with the stack based on hicn light.

For running the hicn-plugin at the server there are two main alternatives:

- Use a docker container
- Run the hicn-plugin directly in a VM or Bare Metal Server

19.3.1 Docker VPP hICN proxy

Install docker in the server VM:

```
server$ curl get.docker.com | bash
```

Run the hicn-http-proxy container. Here we use a public server at `localhost` as origin and HTTP traffic is server with an IPv6 name prefix `b001`.

```
#!/bin/bash

# http proxy options
ORIGIN_ADDRESS=${ORIGIN_ADDRESS:-"localhost"}
ORIGIN_PORT=${ORIGIN_PORT:-"80"}
CACHE_SIZE=${CACHE_SIZE:-"10000"}
DEFAULT_CONTENT_LIFETIME=${DEFAULT_CONTENT_LIFETIME:-"7200"}
HICN_MTU=${HICN_MTU:-"1300"}
FIRST_IPV6_WORD=${FIRST_IPV6_WORD:-"b001"}
USE_MANIFEST=${USE_MANIFEST:-"true"}
HICN_PREFIX=${HICN_PREFIX:-"http://webserver"}

# udp punting
HICN_LISTENER_PORT=${HICN_LISTENER_PORT:-33567}
TAP_ADDRESS_VPP=192.168.0.2
TAP_ADDRESS_KER=192.168.0.1
TAP_ADDRESS_NET=192.168.0.0/24
TAP_ID=0
TAP_NAME=tap${TAP_ID}

vppctl create tap id ${TAP_ID}
vppctl set int state ${TAP_NAME} up
vppctl set interface ip address tap0 ${TAP_ADDRESS_VPP}/24
ip addr add ${TAP_ADDRESS_KER}/24 brd + dev ${TAP_NAME}

# Redirect the udp traffic on port 33567 (The one used for hicn) to vpp
iptables -t nat -A PREROUTING -p udp --dport ${HICN_LISTENER_PORT} -j DNAT \
                   --to-destination ${TAP_ADDRESS_VPP}:${HICN_LISTENER_PORT}
# Masquerade all the traffic coming from vpp
iptables -t nat -A POSTROUTING -j MASQUERADE --src ${TAP_ADDRESS_NET} ! \
                                  --dst ${TAP_ADDRESS_NET} -o eth0
# Add default route to vpp
```

(continues on next page)

(continued from previous page)

```
vppctl ip route add 0.0.0.0/0 via ${TAP_ADDRESS_KER} ${TAP_NAME}
# Set UDP punting
vppctl hicn punting add prefix ${FIRST_IPV6_WORD}::/16 intfc ${TAP_NAME}\
                                type udp4 dst_port ${HICN_LISTENER_PORT}

# Run the http proxy
PARAMS="-a ${ORIGIN_ADDRESS} "
PARAMS+="-p ${ORIGIN_PORT} "
PARAMS+="-c ${CACHE_SIZE} "
PARAMS+="-m ${HICN_MTU} "
PARAMS+="-P ${FIRST_IPV6_WORD} "
PARAMS+="-l ${DEFAULT_CONTENT_LIFETIME} "
if [ "${USE_MANIFEST}" = "true" ]; then
  PARAMS+="-M "
fi

hicn-http-proxy ${PARAMS} ${HICN_PREFIX}
```

Docker images of the example above are available at https://hub.docker.com/r/icnteam/vhttpproxy. Images can be pulled using the following tags.

```
docker pull icnteam/vhttpproxy:amd64
docker pull icnteam/vhttpproxy:arm64
```

Client side

Run the hicn-light forwarder:

```
client$ sudo /usr/bin/hicn-light-daemon --daemon --capacity 1000 --log-file \
                   ${HOME}/hicn-light.log --config ${HOME}/etc/hicn-light.conf
```

Run the http client *higet* and print the http response on stdout:

```
client$ /usr/bin/higet -O - http://webserver/index.html -P c001
```

19.3.2 Host/VM

You can install the hicn-plugin of vpp on your VM and directly use DPDK compatible nics, forwarding hicn packets directly over the network. DPDK compatible nics can be used inside a container as well.

```
server$ sudo apt-get install -y hicn-plugin vpp-plugin-dpdk hicn-apps-memif
```

It will install all the required deps (vpp, hicn apps and libraries compiled for communicating with vpp using shared memories). Configure VPP following the steps described here.

This tutorial assumes you configured two interfaces in your server VM:

- One interface which uses the DPDK driver, to be used by VPP
- One interface which is still owned by the kernel

The DPDK interface will be used for connecting the server with the hicn client, while the other interface will guarantee connectivity to the applications running in the VM, including the hicn-http-proxy. If you run the commands:

```
server$ sudo systemctl restart vpp
server$ vppctl show int
```

The output must show the dpdk interface owned by VPP:

```
              Name               Idx    State  MTU (L3/IP4/IP6/MPLS)     Counter    ␣
↪    Count
GigabitEthernetb/0/0              1      down         9000/0/0/0
local0                            0      down          0/0/0/0
```

If the interface is down, bring it up and assign the correct ip address to it:

```
server$ vppctl set int state GigabitEthernetb/0/0 up
server$ vppctl set interface ip address GigabitEthernetb/0/0 9001::1/64
```

Take care of replacing the interface name (`GigabitEthernetb/0/0`) with the actual name of your interface.

Now enable the hicn plugin and set the punting for the hicn packets:

```
server$ vppctl hicn control start
server$ vppctl hicn punting add prefix c001::/16 intfc GigabitEthernetb/0/0 type ip
```

Run the hicn-http-proxy app:

```
server$ sudo /usr/bin/hicn-http-proxy -a example.com -p 80 -c 10000 -m 1200 -P c001␣
↪http://webserver
```

Configure the client for sending hicn packets without any udp encapsulation:

```
client$ mkdir -p ${HOME}/etc
client$ LOCAL_IP="9001::2"
client$ REMOTE_IP="9001::1"
client$ cat << EOF > ${HOME}/etc/hicn-light.conf
add listener hicn lst 0::0
add punting lst c001::/16
add listener hicn list0 ${LOCAL_IP}
add connection hicn conn0 ${REMOTE_IP} ${LOCAL_IP}
add route conn0 c001::/16 1
EOF
```

Restart the forwarder:

```
client$ sudo killall -INT hicn-light-daemon
client$ sudo /usr/bin/hicn-light-daemon --daemon --capacity 1000 --log-file ${HOME}/
↪hicn-light.log --config ${HOME}/etc/hicn-light.conf
```

Retrieve a web page:

```
client$ /usr/bin/higet -O - http://webserver/index.html -P c001
```

CHAPTER 20

HICN Plugin for Wireshark

The `packethicn` plugin adds support to Wireshark to parse and dissect HICN traffic.

`packethicn` can be compiled and installed in two ways:

1. Alongside HICN, from the HICN root dir (see *Build with HICN*)
2. As a standalone component (see *Standalone build*)

The second one is preferred if HICN is already installed in the system.

CHAPTER 21

Supported platforms

`packethicn` has been tested in

- Ubuntu 20.04
- macOS 12.3

Other platforms and architectures may work.

CHAPTER 22

Installation

22.1 Build with HICN

22.1.1 Dependencies

```
$ sudo add-apt-repository ppa:wireshark-dev/stable

$ sudo apt install -y build-essential cmake wireshark wireshark-dev libgcrypt-dev␣
↪libgnutls28-dev
```

22.1.2 Build and install

From the root HICN dir add the `-DBUILD_WSPLUGIN` flag to cmake.

```
$ cd hicn
$ mkdir build; cd build
$ cmake -DBUILD_APPS=ON -DBUILD_WSPLUGIN=ON ..
$ make -j`nproc`
$ sudo make install
```

22.2 Standalone build

22.2.1 Linux (Ubuntu)

Install dependencies

```
$ sudo add-apt-repository ppa:wireshark-dev/stable
$ curl -s https://packagecloud.io/install/repositories/fdio/release/script.deb.sh |␣
↪sudo bash
$ sudo apt install -y build-essential cmake libhicn-dev wireshark wireshark-dev␣
↪libgcrypt-dev libgnutls28-dev
```

Compile and install HICN wireshark plugin

```
$ cd packethicn
$ mkdir build; cd build
$ cmake ..
$ make
$ sudo make install
```

22.2.2 macOS

If installing wireshark via brew use the `./install_macos.sh` script as shown below:

```
$ brew tap icn-team/hicn-tap
$ brew install hicn
$ brew install wireshark
$ brew install cask wireshark
$ cd packethicn
$ ./install_macos.sh
```

Otherwise (if wireshark was compiled from sources) you can follow the setup for Linux:

```
$ cd packethicn
$ mkdir build; cd build
$ cmake ..
$ make
$ sudo make install
```

CHAPTER 23

Usage

23.1 Filters