hicn Documentation

Release 20.01

Luca Muscariello

Mar 31, 2022

Contents

Hybrid Information-Centric Networking (hICN) is a network architecture that makes use of IPv6 or IPv4 to realize location-independent communications. It is largely inspired by the pioneer work of Van Jacobson on Content-Centric Networking, that was a clean-slate architecture whereas hICN is based on the Internet protocol and easy 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 without replicated unicast. The project implements novel 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 any mobile and desktop operating system.

CHAPTER 1

Getting started

1.1 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.

1.2 Directory layout

Directory name	Description
lib	Core support library
hicn-plugin	VPP plugin
hicn-light	Lightweight packet forwarder
libtransport	Support library with transport layer and API
utils	Tools for testing
apps	Application examples using hicn stack
ctrl	Tools and libraries for network management and control

hicn plugin is a VPP plugin that implement hicn packet processing as specified in https://datatracker.ietf.org/doc/draft-muscariello-intarea-hicn/. 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).

1.3 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.

1.4 Supported platforms

- Ubuntu 18.04 LTS (amd64, arm64)
- Debian Stable/Testing
- Red Hat Enterprise Linux 7
- CentOS 7
- Android 10 (amd64, arm64)
- iOS 13
- macOS 10.15
- Windows 10

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

1.4.1 Ubuntu

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

1.4.2 CentOS

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

1.4.3 macOS

```
brew install hicn
```

1.4.4 Android

Install the applications via the Google Play Store https://play.google.com/store/apps/developer?id=ICN+Team

1.4.5 iOS

Coming soon.

1.4.6 Windows

Coming soon.

1.4.7 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
```

1.4.8 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.

1.5 License

This software is distributed under the following license:

```
Copyright (c) 2017-2020 Cisco and/or its affiliates.
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at:

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

CHAPTER 2

Core library

2.1 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.

2.2 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
```

2.3 Using libhicn

2.3.1 Dependencies

Build dependencies:

- C11 (clang / gcc)
- CMake 3.4

Basic dependencies: None

2.4 Installation

2.4.1 Release mode

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

2.4.2 Debug mode

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

CHAPTER 3

VPP Plugin

3.1 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

3.2 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 make 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
sudo make 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
sudo make 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.

3.3 Using hICN plugin

3.3.1 Dependencies

Build dependencies:

- VPP 20.01
  - DEB packages (can be found https://packagecloud.io/fdio/release/install):
    - vpp
    - libvppinfra-dev
    - vpp-dev

Runtime dependencies:

- VPP 20.01
  - DEB packages (can be found https://packagecloud.io/fdio/release/install):
    - vpp
    - vpp-plugin-core
    - vpp-plugin-dpdk (only to use DPDK compatible nics)

Hardware support (not mandatory):

- DPDK compatible NICs

3.4 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.

3.4.1 Setup the host for VPP

Hugepages must be enabled in the system.

```
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
```

3.4.2 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.

3.4.3 Start VPP

VPP can be started as a process or a service:

Start VPP as a service in Ubuntu 16.04+:

```
sudo systemctl start vpp
```

Start VPP as a process:

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

3.4.4 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/`).

3.4.5 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 `ping_server` application on the host where the forwarder B is running:

```
sudo ping_server -n b002::1
```

Then `ping_client` on the host where forwarder B is running:

```
sudo ping_client -n b002::1
```

3.4.6 Example: packet generator

The packet generator can be used to test the performace 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 0x00000000000000005002000000001f4
    }
}
```

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 4

Transport library

4.1 Introduction

This 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*.
- Connectors for connecting the application to either the hicn-plugin 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)

4.2 Build dependencies

4.2.1 Ubuntu

- libparc
- libmemif (linux only, if compiling with VPP support)
- libasio

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

- DEB packages:
  - vpp

- vpp-lib
- vpp-dev

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

4.2.2 macOS

We recommend to use HomeBrew for installing the libasio dependency:

```
brew install asio
```

Download, compile and install libparc:

```
git clone -b cframework/master https://gerrit.fd.io/r/cicn cframework && cd cframework
mkdir -p libparc.build && cd libparc.build
cmake ../libparc
make
make install
```

Libparc will be installed by default under `/usr/local/lib` and `/usr/local/include`. Since VPP does not support macOS, the hicn-plugin connector is not built.

4.3 Build the library

From the project root folder:

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

4.3.1 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.
- `LIBPARC_HOME`: The folder containing the libparc headers and libraries.
- `VPP_HOME`: The folder containing the installation of VPP.
- `LIBMEMIF_HOME`: The folder containing the libmemif headers and libraries.
- `BUILD_MEMIF_CONNECTOR`: On linux, set this value to `ON` for building the VPP connector.

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

4.3.2 Install the library

For installing the library, from the cmake build folder:

```
sudo make install
```

CHAPTER 5

Portable forwarder

5.1 Introduction

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

5.2 Using hicn-light

5.2.1 Dependencies

Build dependencies:

- C99 (clang / gcc)
- CMake 3.4

Basic dependencies:

- OpenSSL
- pthreads
- libevent
- libparc

5.3 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.

5.3.1 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␣
↪facility=level]
                  [--log-file filename] [--config file]

Options:
--port <tcp_port>           = tcp port for local in-bound connections
--daemon                    = start as daemon process
--capacity <capacity>       = 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 a facility to a given log level. You can have␣
↪multiple of these.
                              facilities: all, config, core, io, message, processor
                              levels: debug, info, notice, warning, error, critical,␣
↪alert, off
                              example: hicn-light-daemon --log io=debug --log core=off
--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. When a configuration file is specified, no default listeners are setup. Only 'add listener' lines in the configuration file matter.

If no configuration file is specified, 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. Commands are expected on port 2001.

5.3.2 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.

Information about the commands are also available in the `hicn-light-control` help message.

`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_adress> <local_port>

  <symbolic>        :User defined name for listener, must start with alpha and␣
↪bealphanum
  <protocol>        :tcp | udp
  <localAddress>    :IPv4 or IPv6 address
  <local_port>      :TCP/UDP port
```

`add listener hicn`: creates a hicn listener with the specified options on the local forwarder.

```
add listener hicn <symbolic> <local_adress>

  <symbolic>        :User defined name for listener, must start with alpha and be␣
↪alphanum
  <localAddress>    :IPv4 or IPv6 address
```

`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
```

`add connection hicn`: creates an hicn connection on the local forwarder with the specified options.

```
add connection hicn <symbolic> <remote_ip> <local_ip>

  <symbolic>            : symbolic name, e.g. 'conn1' (must be unique, start with␣
↪alpha)
  <remote_ip>           : the IPv4 or IPv6 of the remote system
  <local_ip>            : local IP address to bind to
```

`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 'help list connections')
  <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 <protocol> <symbolic | connid>

```

(continues on next page)

(continued from previous page)

```
  <protocol>   : tcp | upd. This is the protocol used to create the connection.
  <symbolic>   :The symbolic name for an exgress (must be unique, start with alpha)
  <connid>:    :The egress connection id (see 'help list connections')
```

`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
```

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

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

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

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

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

```
cache clear
```

`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.

```
set strategy <prefix> <strategy>

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

`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')
```

`add punting`: add punting rules to the forwarders.

```
add punting <symbolic> <prefix>

 <symbolic> : listener symbolic name
 <address>  : prefix to add as a punting rule. (example 1234::0/64)
```

`mapme enable`: enables/disables mapme.

```
mapme enable <on|off>
```

`mapme discovery`: enables/disables mapme discovery.

```
mapme discovery <on|off>
```

`mapme timescale`: set the timescale value expressed in milliseconds.

```
mapme timescale <milliseconds>
```

`mapme retx`: set the retransmission time value expressed in millisecond.

```
mapme retx <milliseconds>
```

`quit`: exits the interactive bash.

5.3.3 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

#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

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

CHAPTER 6

Face manager

6.1 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.

6.1.1 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.

6.1.2 Architectural overview

Facelets

TODO:

```
- Key attributes (netdevice and protocol family)
- Facelet API
```

Events

TODO

Facelet cache & event scheduling

TODO:

```
 - Facelet cache
 - Joins
 - How synchronization work
```

6.1.3 Interface API

TODO

6.2 Developing a new interface

6.2.1 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);
```

(continues on next page)

(continued from previous page)

```
    /* 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 instanciation 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.

TODO: detail the structure of the file

6.2.2 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 7

Control plane support

Control plane functionalities are provides via SDN controllers or via standard IP routing protocols. SDN support is provided by using the NETCONF/YANG protocol for network management, control and telemetry.

Routing is supported via synchronization of the IP FIB and the IP RIB as implemented by one of the routing protocols in FRR. Without loss of generality we have reported below one example of IGP routing via OSPF for IPv6.

The VPP IP FIB can be controlled and updated by one FRR routing protocol which is used for routing over locators and also over hICN name prefixes.

7.1 NETCONF/YANG

7.1.1 Getting started

NETCONF/YANG support is provided via several external components such as libyang, sysrepo, libnetconf and netopeer. The hicn project provides a sysrepo plugin and a YANG model for two devices: the VPP based hicn virtual switch and the portable forwarder. The YANG model for the VPP based hICN vSwitch is based the full hICN C API exported by the VPP plugin with the addition of some VPP APIs such as interface and FIB management which are required by the hICN plugin.

To install libyang, sysrepo, libnetconf and netopeer2 for Ubuntu18 amd64/arm64 or CentOS 7 and ad-hoc repository is available and maintained in bintray at https://dl.bintray.com/icn-team/apt-hicn-extras.

For instance in Ubuntu 18 LTS:

Install the sysrepo YANG data store and a NETCONF server:

```
echo "deb [trusted=yes] https://dl.bintray.com/icn-team/apt-hicn-extras bionic main" \
                                              | tee -a /etc/apt/sources.list
apt-get update && apt-get install -y libyang sysrepo libnetconf2 netopeer2-server
```

Install the VPP based hICN virtual switch:

```
curl -s https://packagecloud.io/install/repositories/fdio/release/script.deb.sh | bash
apt-get update && apt-get install -y hicn-plugin vpp-plugin-dpdk hicn-sysrepo-plugin
```

The hICN YANG models are installed under `/usr/lib/$(uname -m)-linux-gnu/modules_yang`.

Configure the NETCONF/YANG components:

```
bash /usr/bin/setup.sh sysrepoctl /usr/lib/$(uname -m)-linux-gnu/modules_yang root
bash /usr/bin/merge_hostkey.sh sysrepocfg openssl
bash /usr/bin/merge_config.sh sysrepocfg genkey
```

You can manually install the yang model using the following bash script:

```
EXIT_CODE=0
command -v sysrepoctl > /dev/null
if [ $? != 0 ]; then
    echo "Could not find command \"sysrepoctl\"."
     exit ${EXIT_CODE}
else
sysrepoctl --install --yang=path_to_hicn_yang_model
fi
```

7.1.2 YANG model

hicn.yang can be found in the yang-model. It consists of two container nodes:

```
|--+ hicn-conf: holds the configuration data;
|  |--+ params: contains all configuration parameters;
|--+ hicn-state: provides the state data
|  |--+ state,
|  |--+ strategy,
|  |--+ strategies,
|  |--+ route,
|  |--+ face-ip-params
and corresponding leaves.
```

A controller can configure these parameters through the edit-config RPC call. This node can be used to enable and to initialize the hicn-plugin in VPP instance. hicn-state container is used to provide the state data to the controller. It consists of state, strategy, strategies, route, and face-ip-params nodes with the corresponding leaves. In the hicn model a variety of RPCs are provided to allow controller to communicate with the hicn-plugin as well as update the state data in hicn-state.

7.1.3 Example

To setup the startup configuration you can use the following script:

```
EXIT_CODE=0
command -v sysrepocfg > /dev/null
if [ $? != 0 ]; then
    echo "Could not find command \"sysrepocfg\"."
     exit ${EXIT_CODE}
else
sysrepocfg -d startup -i path_to_startup_xml -f xml hicn
fi
```

startup.xml is placed in the yang-model. Here you can find the content:

```
<hicn-conf  xmlns="urn:sysrepo:hicn">
<params>
    <enable_disable>false</enable_disable>
    <pit_max_size>-1</pit_max_size>
    <cs_max_size>-1</cs_max_size>
    <cs_reserved_app>-1</cs_reserved_app>
    <pit_dflt_lifetime_sec>-1</pit_dflt_lifetime_sec>
    <pit_max_lifetime_sec>-1</pit_max_lifetime_sec>
    <pit_min_lifetime_sec>-1</pit_min_lifetime_sec>
</params>
</hicn-conf>
```

It contains the leaves of the parameters in hicn-conf node which is used as the startup configuration. This configuration can be changed through the controller by subscribing which changes the target to the running state. hicn yang model provides a list of RPCs which allows controller to communicate directly with the hicn-plugin. This RPCs may also cause the modification in state data.

In order to run different RPCs from controller you can use the examples in the controler_rpcs_instances.xml in the yang-model. Here you can find the content:

```
<node-params-get xmlns="urn:sysrepo:hicn"/>

<node-stat-get xmlns="urn:sysrepo:hicn"/>

<strategy-get xmlns="urn:sysrepo:hicn">
    <strategy_id>0</strategy_id>
</strategy-get>

<strategies-get xmlns="urn:sysrepo:hicn"/>

<route-get xmlns="urn:sysrepo:hicn">
    <prefix0>10</prefix0>
    <prefix1>20</prefix1>
    <len>30</len>
</route-get>

<face-params-get xmlns="urn:sysrepo:hicn">
    <faceid>10</faceid>
</face-params-get>

<hicn-enable xmlns="urn:sysrepo:hicn">
    <prefix>b001::/64</prefix>
</hicn-enable>

<hicn-disable xmlns="urn:sysrepo:hicn">
    <prefix>b001::/64</prefix>
</hicn-disable>
```

Run the plugin

First, verify the plugin and binary libraries are located correctly, then run the vpp through (service vpp start). Next, run the sysrepo plugin (sysrepo-plugind), for debug mode: sysrep-plugind -d -v 4 which runs with high verbosity. Now, the hicn sysrepo plugin is loaded. Then, run the netopeer2-server which serves as NETCONF server

Connect from netopeer2-cli

In order to connect through the netopeer client run the netopeer2-cli. Then, follow these steps:

- connect –host XXX –login XXX
- get (you can get the configuration and operational data)
- get-config (you can get the configuration data)
- edit-config –target running –config

With the default netopeer2-server configuration the authentication required by netopeer2-cli reflects the ssh authentication (username and password or public key). For other means of authentication please refer to netopeer2-server documentation (e.g., netopeer2/server/configuration/README.md).

You can modify the configuration but it needs an xml configuration input.

```
<hicn-conf  xmlns="urn:sysrepo:hicn">
<params>
    <enable_disable>false</enable_disable>
    <pit_max_size>-1</pit_max_size>
    <cs_max_size>-1</cs_max_size>
    <cs_reserved_app>-1</cs_reserved_app>
    <pit_dflt_lifetime_sec>-1</pit_dflt_lifetime_sec>
    <pit_max_lifetime_sec>-1</pit_max_lifetime_sec>
    <pit_min_lifetime_sec>-1</pit_min_lifetime_sec>
</params>
</hicn-conf>
```

- user-rpc (you can call one of the rpc proposed by hicn model but it needs an xml input)

Connect from OpenDaylight (ODL) controller

In order to connect through the OpenDaylight follow these procedure:

- run karaf distribution (./opendayligh_installation_folder/bin/karaf)
- install the required feature list in DOL (feature:install odl-netconf-server odl-netconf-connector odl-restconf-all odl-netconf-topology or odl-netconf-clustered-topology)
- run a rest client program (e.g., postman or RESTClient)
- mount the remote netopeer2-server to the OpenDaylight by the following REST API:

```
PUT <http://localhost:8181/restconf/config/network-topology:network-topology/
↪topology/topology-netconf/node/hicn-node>`
```

with the following body:

```
<node xmlns="urn:TBD:params:xml:ns:yang:network-topology">
  <node-id>hicn-node</node-id>
  <host xmlns="urn:opendaylight:netconf-node-topology">Remote_NETCONF_SERVER_IP</
↪host>
  <port xmlns="urn:opendaylight:netconf-node-topology">830</port>
  <username xmlns="urn:opendaylight:netconf-node-topology">username</username>
  <password xmlns="urn:opendaylight:netconf-node-topology">password</password>
  <tcp-only xmlns="urn:opendaylight:netconf-node-topology">false</tcp-only>
```

(continues on next page)

(continued from previous page)

```
  <keepalive-delay xmlns="urn:opendaylight:netconf-node-topology">1</keepalive-
↪delay>
</node>
```

Note that the header files must be set to `Content-Type: application/xml`, `Accept: application/xml`.

- send the operation through the following REST API:

POST http://localhost:8181/restconf/operations/network-topology:network-topology/topology/topology-netconf/node/hicn-node/yang-ext:mount/ietf-netconf:edit-config

The body can be used the same as edit-config in netopeer2-cli.

Connect from Cisco Network Services Orchestrator (NSO)

To connect NSO to the netopeer2-server, first, you need to write a NED package for your device. The procedure to create NED for hicn is explained in the following:

Place hicn.yang model in a folder called hicn-yang-model, and follow these steps:

- ncs-make-package –netconf-ned ./hicn-yang-model ./hicn-nso
- cd hicn-nso/src; make
- ncs-setup –ned-package ./hicn-nso –dest ./hicn-nso-project
- cd hicn-nso-project
- ncs
- ncs_cli -C -u admin
- configure
- devices authgroups group authhicn default-map remote-name user_name remote-password password
- devices device hicn address IP_device port 830 authgroup authhicn device-type netconf
- state admin-state unlocked
- commit
- ssh fetch-host-keys

At this point, we are able to connect to the remote device.

7.2 Release note

The current version is compatible with the 20.01 VPP stable and sysrepo devel.

7.3 Routing plugin for VPP and FRRouting for OSPF6

This document describes how to configure the VPP with hicn_router plugin and FRR to enable the OSPF protocol. The VPP and FRR are configured in a docker file.

7.3.1 DPDK configuration on host machine

Install and configure DPDK:

```
make install T=x86_64-native-linux-gcc && cd x86_64-native-linux-gcc && sudo make␣
↪install
modprobe uio
modprobe uio_pci_generic
dpdk-devbind --status
the PCIe number of the desired device can be observed ("xxx")
sudo dpdk-devbind -b uio_pci_generic "xxx"
```

7.3.2 VPP configuration

Run and configure the VPP (hICN router plugin is required to be installed in VPP):

```
vpp# set int state TenGigabitEtherneta/0/0 up
vpp# set int ip address TenGigabitEtherneta/0/0 a001::1/24
vpp# create loopback interface
vpp# set interface state loop0 up
vpp# set interface ip address loop0 b001::1/128
vpp# enable tap-inject  # This creates the taps by router plugin
vpp# show tap-inject # This shows the created taps
vpp# ip mroute add ff02::/64 via local Forward  # ff02:: is multicast ip address
vpp# ip mroute add ff02::/64 via TenGigabitEtherneta/0/0 Accept
vpp# ip mroute add ff02::/64 via loop0 Accept
```

Setup the tap interface:

```
ip addr add a001::1/24 dev vpp0
ip addr add b001::1/128 dev vpp1
ip link set dev vpp0 up
ip link set dev vpp1 up
```

7.3.3 FRR configuration

Install FRR in Ubuntu 18 LTS: http://docs.frrouting.org/projects/dev-guide/en/latest/building-frr-for-ubuntu1804.html

Run and configure FRRouting (ospf):

```
/usr/lib/frr/frrinit.sh start &
vtysh
configure terminal
router ospf6
area 0.0.0.0 range a001::1/24
area 0.0.0.0 range b001::1/128
interface vpp0 area 0.0.0.0
interface vpp1 area 0.0.0.0
end
wr
add  "no ipv6 nd suppress-ra" to the first configurtion part of the /etc/frr/frr.conf
```

After the following configuration, the traffic over tap interface can be observed via `tcpdump- i vpp1`. The neighborhood and route can be seen with the `show ipv6 ospf6 neighbor/route` command.

CHAPTER 8

Telemetry

Tools to collect telemetry from hICN forwarders.

8.1 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.

8.2 Quick start

From the code tree root:

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

8.3 Using hICN collectd plugins

8.3.1 Platforms

hICN collectd plugins have been tested in:

- Ubuntu 16.04 LTS (x86_64)
- Ubuntu 18.04 LTS (x86_64)
- Debian Stable/Testing
- Red Hat Enterprise Linux 7
- CentOS 7

8.3.2 Dependencies

Build dependencies:

- VPP 20.01, 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`

8.4 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).

8.5 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.

8.5.1 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 9

Utility applications

9.1 Introduction

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

9.2 Using hICN utils applications

9.2.1 Dependencies

Build dependencies:

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

Basic dependencies:

- OpenSSL
- pthreads
- libevent
- libparc
- libhicntransport

9.3 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).

9.3.1 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)
-V                          = verbose, prints statistics about the messagges sent and
↪received (default false)
-D                          = dump, dumps sent and received packets (default false)
-q                          = quiet, no printing (default false)
-d                          = daemon mode
-H                          = help
```

Example:

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

9.3.2 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)
-V                          = verbose, prints statistics about the messagges sent and
↪received (default false)
-D                          = dump, dumps sent and received packets (default false)
-q                          = quiet, no printing (default false)
-H                          = help
```

Example:

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

9.3.3 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:

```
usage: hiperf [-S|-C] [options] [prefix|name]

SERVER OR CLIENT:
-D                          = Run as a daemon
-R                          = Run RTC protocol (client or server)
-f <filename>               = Log file

SERVER SPECIFIC:
-A <content_size>           = Size of the content to publish. This is not the size␣
↪of the packet (see -s for it).
-s <packet_size>            = Size of the payload of each data packet.
-r                          = Produce real content of <content_size> bytes
-m                          = 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
-y <hash_algorithm>         = Use the selected hash algorithm for calculating␣
↪manifest digests
-p <password>               = Password for p12 keystore
-x                          = Produce a content of <content_size>, then after␣
↪downloading it produce a new content of <content_size> without resetting the suffix␣
↪to 0.
-B <bitrate>                = Bitrate for RTC producer, 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
-E                          = Enable encrypted communication. Requires the path to a␣
↪p12 file containing the crypto material used for the TLS handshake

CLIENT SPECIFIC:
-b <beta_parameter>         = RAAQM beta parameter
-d <drop_factor_parameter>  = RAAQM drop factor parameter
-L <interest lifetime>      = Set interest lifetime.
-M <Download for real>      = Store the content downloaded.
-W <window_size>            = Use a fixed congestion window for retrieving the data.
-i <stats_interval>         = Show the statistics every <stats_interval>␣
↪milliseconds.
-v                          = Enable verification of received data
-c <certificate_path>       = Path of the producer certificate to be used for␣
↪verifying the origin of the packets received. Must be used with -v.
-k <passphrase>             = String from which is derived the symmetric key used by␣
↪the producer to sign packets and by the consumer to verify them. Must be used with -
↪v.
-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
```

Example:

```
hiperf -S c001::/64
```

9.4 Client/Server benchmarking using `hiperf`

9.4.1 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 \
                   libparc-dev
mkdir hicn-suite && cd hicn-suite
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=../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
```

9.4.2 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 \
                   libparc-dev
mkdir hicn-suite && cd hicn-suite
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 10

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.

10.1 Dependencies

Build dependencies:

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

Basic dependencies:

- OpenSSL
- pthreads
- libevent
- libparc
- libcurl
- libhicntransport

10.2 Executables

10.2.1 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.

10.2.2 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*.

10.3 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

10.3.1 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
```

10.3.2 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
```