Polycube Documentation

The Polycube Authors

Sep 10, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION TO POLYCUBE

Polycube is an **open source** software framework for Linux that provides **fast** and **lightweight network functions**, such as *bridge*, *router*, *nat*, *load balancer*, *firewall*, *DDoS mitigator*, and more.

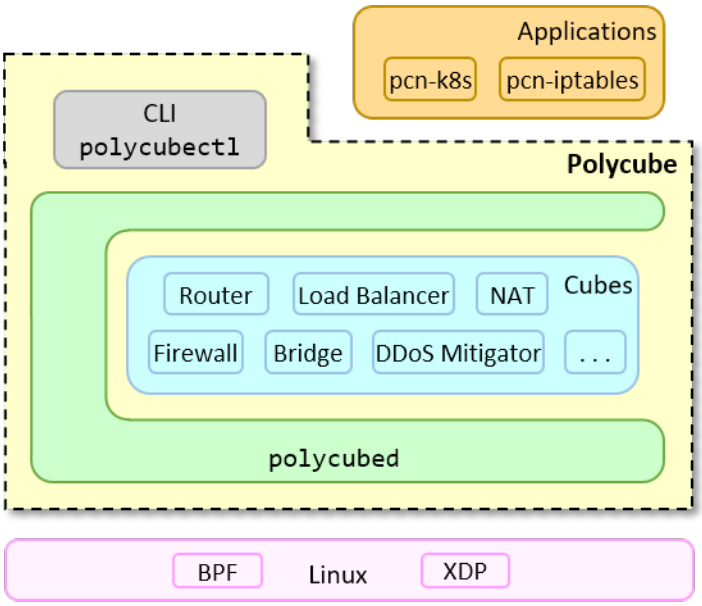
Individual network functions can be composed to build arbitrary **service chains** and provide custom network connectivity to **namespaces**, **containers**, **virtual machines**, and **physical hosts**. Polycube supports also multitenancy, with multiple **virtual networks** that can be enabled concurrently.

Network functions, called *cubes*, are extremely **efficient** because are based on the recent *BPF* and *XDP* Linux kernel technologies. In addition, cubes are easily **extensible** and **customizable**.

Polycube can control its entire virtual topology and all the network services with a simple and coherent command line, available through the `polycubectl` tool. A set of equivalent commands can be issued diretly to `polycubed`, the Polycube REST-based daemon, for better machine-to-machine interaction.

Polycube also provides two working **standalone applications** built up using this framework. **pcn-k8s** is a Polycube-based CNI plug-in for *Kubernetes*, which can handle the network of an entire data center. It also delivers better throughput as compared with some of the existing CNI plug-ins. **pcn-iptables** is a more efficient and scalable clone of the existing Linux *iptables*.

A brief overview of the Polycube layered structure, including the command line interface (CLI), standalone applications, and some of the available cubes, is shown in the picture below.



1.1 Main features

1.1.1 Extremely fast

Polycube enables extremely fast and efficient network services, thanks to its capability to run inside the Linux kernel and, whenever possible, as close as possible to the network interface card driver, which reduces the time spent in ancillary components.

1.1.2 Designed with service chaining in mind

Polycube supports the definition of multiple network scenarios through the composition of many elementary building blocks (i.e., *cubes*), which can be combined (e.g., attached one to the other) to create complex network services. For instance, several dockers can communicate through a bridge, which is then attached to a router to provide internet connectivity (possibly through a nat), while a firewall protects the entire infrastructure.

Polycube has been designed to simplify service chaining: cubes can be dynamically instantiated and seamlessly connected together using virtual links, mimicking traditional networks in which dedicated middlebox are connected with each other through physical wires. As a consequence, cubes can be composed to build arbitrary service chains and provide custom network connectivity to namespaces, docker, virtual machines, and physical hosts.

1.1.3 Production-grade network services

Polycube greatly simplifies the communication between data, control, and management planes of a network service, hence it enables the creation of rich network services which include all the above features. In turn, this offers a simplified environment to service developers, who can leverage the power of Polycube to write the data/control and management parts of their network services.

In detail, most network services include a *data plane*, such as the longest prefix match algorithm in a router, a *control plane*, e.g., where routing protocols are executed, and a *management plane*, devoted to the configuration and monitoring of the service. *bpf* aims at the creation of fast data planes, leaving the rest under the responsibility of the developer; Polycube overcomes this limitation with a rich set of primitives natively provided by the framework. In addition, Polycube provides the software infrastructure required to overcome possible limitations of BPF in the data plane (e.g., the limited size of data plane programs), enabling the steering of packets that require complex processing in user-space, where previous limitations do not apply.

1.1.4 Single point of control

Polycube provides a single point of control to the entire virtual network, including all the running services. Its unified command line interface enables the *setup* of the virtual infrastructure, it *instantiates* new services and connect them properly, handles the *lifecycle* of all running cubes, and supports the *configuration* and *monitoring* of all the running elements.

This is achieved by a unified command line interface (*polycubectl*) that interacts with a REST-based daemon (*polycubed*) in charge of the supervision of the entire infrastructure. In addition, Polycube implements a *service agnostic* configuration mechanism, based on YANG data models and the RESTCONF protocol, in which new services can seamlessly develop and dynamically added to the framework, with the command line being automatically able to handle the above services without any modification.

1.1.5 Outstanding performance with real applications

Two standalone applications have been released to show the potential of Polycube, `pcn-iptables` and `pcn-k8s`.

- **pcn-iptables**: is a clone of **iptables** that is able to filter packets passing through a Linux host, demonstrating how packet filtering can be achieved with impressive performance, while at the same time guaranteeing the same command line and the same external behavior of the original software.
- **pcn-k8s**: is a CNI network plugin for **Kubernetes**, i.e., a software that handles the entire virtual network of a Kubernetes cluster, which includes bridging, routing, NAT, load balancing and tunneling services. Our plug-in has been tested for scalability and guarantees outstanding performance in terms of network throughput.

1.1.6 Powered by eBPF and XDP

BPF and *XDP* are the main Linux kernel technologies on which Polycube is based upon. *BPF* supports dynamic code injection in the Linux kernel at runtime, enabling the dynamic creation of a data plane. The *BPF* data plane has a minimal feature set which avoids processing overhead and is exactly tailored to user needs.

- *bpf* (Extended Berkeley Packet Filter) code is dynamically compiled and injected, checked for safety to avoid any hazard in the kernel, while efficiency is achieved thanks to a just-in-time compiler (JIT) that transforms each instruction into a native 64-bit (x64) code for maximum performance.
- *XDP* (eXpress Data Path) provides a new way to intercept network packets very early in Linux network stack, with a significant gain in performance thanks to the possibility to avoid costly operations such as *skbuff* handling.

1.1.7 Integrated with common Linux tools

While configuring network services, people are already familiar with well-known Linux tools, such as *ifconfig*, *route*, *tcpdump*, and more. To foster a broader integration of Polycube and Linux, Polycube services can be configured with either its native CLI (and REST API), or exploiting most of the networking tools that are already used nowadays.

This offers an easy way for new users to play with Polycube services; it increases the potential of the framework that can leverage tons of existing software (e.g., Quagga for dynamic routing); it enables to seamlessly extend Linux networking with powerful and efficient eBPF/XDP-based services.

CHAPTER

TWO

QUICK START

If you want to try Polycube, this is a five minutes document that will guide you through the basics.

In order to try Polycube you need to use two components:

- `polycubed`: the *Polycube daemon*, that must be up and running.
- `polycubectl`: the *Polycube CLI*, used to interact with the framework.

There are two modes to try Polycube: using *Docker* or installing it on the *Bare Metal*.

2.1 Docker

Docker is the easiest and fastest way to try Polycube because it only requires a recent kernel (see *Updating Linux kernel* for more information).

You can use Polycube by pulling the proper Docker image, either the **most recent released version (stable)** or the **most recent snapshot (from GitHub)** (note that the above commands may require root privileges, i.e., sudo):

- **Released version** (stable):

```
docker pull polycubenets/polycube:v0.9.0-rc
```

- **Most recent snapshot** (from GitHub)

```
docker pull polycubenets/polycube:latest
```

Run the Polycube Docker and launch `polycubed` (the polycube daemon) inside it, as follows:

- **Released version** (stable):

```
docker run  -it --rm --privileged --network host \
-v /lib/modules:/lib/modules:ro -v /usr/src:/usr/src:ro -v /etc/localtime:/etc/
↪localtime:ro \
polycubenets/polycube:v0.9.0-rc /bin/bash -c 'polycubed -d && /bin/bash'
```

- **Most recent snapshot** (from GitHub)

```
docker run  -it --rm --privileged --network host \
-v /lib/modules:/lib/modules:ro -v /usr/src:/usr/src:ro -v /etc/localtime:/etc/
↪localtime:ro \
polycubenets/polycube:latest /bin/bash -c 'polycubed -d && /bin/bash'
```

The Docker container is launched in the host networking stack (`--network host`) and in privileged mode (`--privileged`), which are required to use `eBPF` features. The above command starts `polycubed` (with standard flags) and opens a shell, which can be used to launch `polycubectl`. Refer to *polycubectl CLI* for more information about the CLI interface.

```
polycubectl --help
```

To stop the daemon just remove the container, e.g. exiting from the shell with the `exit` command.

2.2 Bare Metal

This is a more elaborated way, recommended either for *advanced users* or for who would like to use Polycube in production (*not for testing*). Please refer to the *installation guide* to get detailed information about how to compile and install Polycube and its dependencies.

Once you have Polycube installed in your system, you can start the `polycubed` service:

```
# start polycubed service
# (sudo service start polycubed will work in many distros as well)
sudo systemctl start polycubed

# check service status
sudo systemctl status polycubed
```

Start interacting with the framework by using `polycubectl`. Refer to *polycubectl CLI* for more information.

```
polycubectl --help
```

To stop the `polycubed` service use:

```
sudo systemctl stop polycubed
```

CHAPTER

THREE

INSTALLING POLYCUBE FROM SOURCES

This installation guide includes instructions to install Polycube on `Ubuntu >= 18.04`. However those should also work on other versions and distributions with a few changes.

3.1 Dependencies

Polycube requires following dependencies:

- **Recent Linux kernel**: most Polycube services work with kernel **>= v4.15**; however, there are cases in a newer kernel may be required. In case you are unsure, please upgrade to **kernel v5.4** (section *Updating Linux kernel*).
- **pistache**: a library to build rest API servers
- **libints**: a library for crafting packets (needed only for some services)
- **Go language**: required to run `polycubectl` (polycube command line interface)

Following sections will detail the installation process for the above components.

3.2 Updating Linux kernel

Most Polycube services require at least the **Linux kernel v4.15**. However, the *Dynmon service* allows the dynamic injection of a custom data plane, which may exploit latest eBPF kernel features and hence require more up-to-date kernel versions. Therefore we suggest to upgrade the to the latest Linux kernel in order to be on the safe side.

The following examples show how to update kernel to either version **4.15** and **5.4**. To check your current kernel version, please use `uname -a`. After a kernel update, please remember to reboot your machine at choose the newly installed one while the boot process starts.

To update to kernel **v4.15**:

```
wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v4.15/linux-headers-4.15.0-041500_4.
↪15.0-041500.201802011154_all.deb
wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v4.15/linux-headers-4.15.0-041500-
↪generic_4.15.0-041500.201802011154_amd64.deb
wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v4.15/linux-image-4.15.0-041500-
↪generic_4.15.0-041500.201802011154_amd64.deb

sudo dpkg -i *.deb
sudo reboot
```

To update to kernel **v5.4**:

```
wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v5.4/linux-headers-5.4.0-050400_5.4.0-
↪050400.201911242031_all.deb
wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v5.4/linux-headers-5.4.0-050400-
↪generic_5.4.0-050400.201911242031_amd64.deb
wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v5.4/linux-image-5.4.0-050400-generic_
↪5.4.0-050400.201911242031_amd64.deb
wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v5.4/linux-modules-5.4.0-050400-
↪generic_5.4.0-050400.201911242031_amd64.deb

sudo dpkg -i *.deb
sudo reboot
```

To update to **other** kernels:

You can follow the instructions at Upgrading Ubuntu 20.04 to the latest Linux kernel, and possibly adapting the script there. The list of available kernels is available at https://kernel.ubuntu.com/~kernel-ppa/mainline/ .

3.3 Automatic installation from sources

If you are running `Ubuntu >= 18.04` and you do not want to manually install Polycube and its dependencies, you can use the install script available under the `scripts` folder as follows: `./scripts/install.sh`. **DO NOT** run with sudo privileges, it will automatically ask you the password when needed, but the entire operations needs to be run by a default user. The scripts has been tested on `Ubuntu 18.04`, `Ubuntu 19.04` and `Ubuntu 20.04`.

Please notice that this script does not update the kernel version.

In order to install Polycube with the script, you can either download the most recent *Released version (stable)* or compile the *Most recent snapshot (from GitHub)*.

Once the installation is completed, you can follow the *Quick Start* instructions.

Note: if you have llvm 6.0 installed (check with `apt list --installed | grep "llvm"`), the installation script will fail. In this case, remove llvm 6.0 before starting the installation script:

```
sudo apt remove llvm-6.0 llvm-6.0-dev llvm-6.0-runtime
```

3.3.1 Released version (stable)

```
# Download the source pack from Polycube repository
#  (i.e., https://github.com/polycube-network/polycube/releases)
# Look at the most recent version and update the following lines accordingly
wget https://github.com/polycube-network/polycube/releases/download/v0.9.0/polycube-v0.9.
↪0.zip

# Unpack the source files (e.g., this refers to version 0.9)
unzip polycube-v0.9.0.zip

# Move into the newly created folder:
cd polycube-v0.9.0

# Launch the automatic install script *from the Polycube root folder*
```

(continues on next page)

(continued from previous page)

```
#   (use -h to see different installation modes)
./scripts/install.sh
```

3.3.2 Most recent snapshot (from GitHub)

```
# Install git
sudo apt-get install git

# Clone the Polycube repository including all the submodules
git clone https://github.com/polycube-network/polycube
cd polycube
git submodule update --init --recursive

# Launch the automatic install script *from the Polycube root folder*
#   (use -h to see different installation modes)
./scripts/install.sh
```

3.4 Manual installation from the most recent snapshop (on GitHub)

This procedure is discouraged, as the *Automatic installation from sources* looks appropriate for most of the people.

The following steps are required only if you want to compile and install everything manually, without the provides install script (`./scripts/install.sh`).

3.4.1 Install GO

Go 1.8+ is needed to run `polycubectl`, if you only plan to install `polycubed` you can skip this step.

Since Ubuntu 20.04, support for `golang-go` has been introduced into the main repositories. Thus, you just need to run

```
sudo apt install golang-go
```

Instead, for all the previous versions (< 20.04) please refer to the following instructions:

```
# If you are running a previous Ubuntu version, you could add the
# longsleep/golang-backports ppa repository to get get required golang version.
# sudo add-apt-repository ppa:longsleep/golang-backports
# sudo apt-get update
sudo apt-get install golang-go

# Set $GOPATH, if not already set
mkdir $HOME/go
export GOPATH=$HOME/go

# Check the Go version; you should get something
# like 'go version go1.8.3 linux/amd64'
go version

# In order to make permanent the above changes, you can append export commands
```

(continues on next page)

(continued from previous page)

```
# to `~/.bashrc` or run the following commands and restart the terminal.
echo 'export GOPATH=$HOME/go' >> ~/.bashrc
```

3.4.2 Install dependencies

```
# Install Polycube dependencies
sudo apt-get -y install git build-essential cmake bison flex \
    libelf-dev libllvm5.0 llvm-5.0-dev libclang-5.0-dev libpcap-dev \
    libnl-route-3-dev libnl-genl-3-dev uuid-dev pkg-config \
    autoconf libtool m4 automake libssl-dev kmod jq bash-completion \
    gnupg2
```

Install libyang-dev

```
wget -nv http://download.opensuse.org/repositories/home:/liberouter/xUbuntu_18.04/amd64/
↪libyang_0.14.81_amd64.deb -O libyang.deb
wget -nv http://download.opensuse.org/repositories/home:/liberouter/xUbuntu_18.04/amd64/
↪libyang-dev_0.14.81_amd64.deb -O libyang-dev.deb
sudo apt install -f ./libyang.deb
sudo apt install -y -f ./libyang-dev.deb
rm ./libyang.deb
rm ./libyang-dev.deb
```

If you are using another operating system please check the libyang installation documentation.

Install pistache

```
# Install Pistache (a library to create web servers that is used in polycubed)
git clone https://github.com/oktal/pistache.git
cd pistache
# use last known working version
git checkout 117db02eda9d63935193ad98be813987f6c32b33
git submodule update --init
mkdir build &&  cd build
cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release -DPISTACHE_USE_SSL=ON ..
make -j $(getconf _NPROCESSORS_ONLN)
sudo make install
```

Install libtins

```
# Install libtins (a library for network packet sniffing and crafting, used to create␣
↪packets)
git clone --branch v3.5 https://github.com/mfontanini/libtins.git
cd libtins
mkdir build && cd build
cmake -DLIBTINS_ENABLE_CXX11=ON -DLIBTINS_BUILD_EXAMPLES=OFF \
    -DLIBTINS_BUILD_TESTS=OFF -DLIBTINS_ENABLE_DOT11=OFF \
    -DLIBTINS_ENABLE_PCAP=OFF -DLIBTINS_ENABLE_WPA2=OFF \
    -DLIBTINS_ENABLE_WPA2_CALLBACKS=OFF ..
make -j $(getconf _NPROCESSORS_ONLN)
sudo make install
sudo ldconfig
```

3.4.3 Installing Polycube

This installs the Polycube daemon (`polycubed`), the Polycube CLI (`polycubectl`) and the network services shipped with Polycube. If you want to disable some services, you can modify the cmake flags using `ccmake`.

```
git clone https://github.com/polycube-network/polycube
cd polycube
git submodule update --init --recursive
mkdir build && cd build
# use 'ccmake ..' to change different compilation options as
# enable/disable some services for example
cmake ..
make -j $(getconf _NPROCESSORS_ONLN)
sudo make install
```

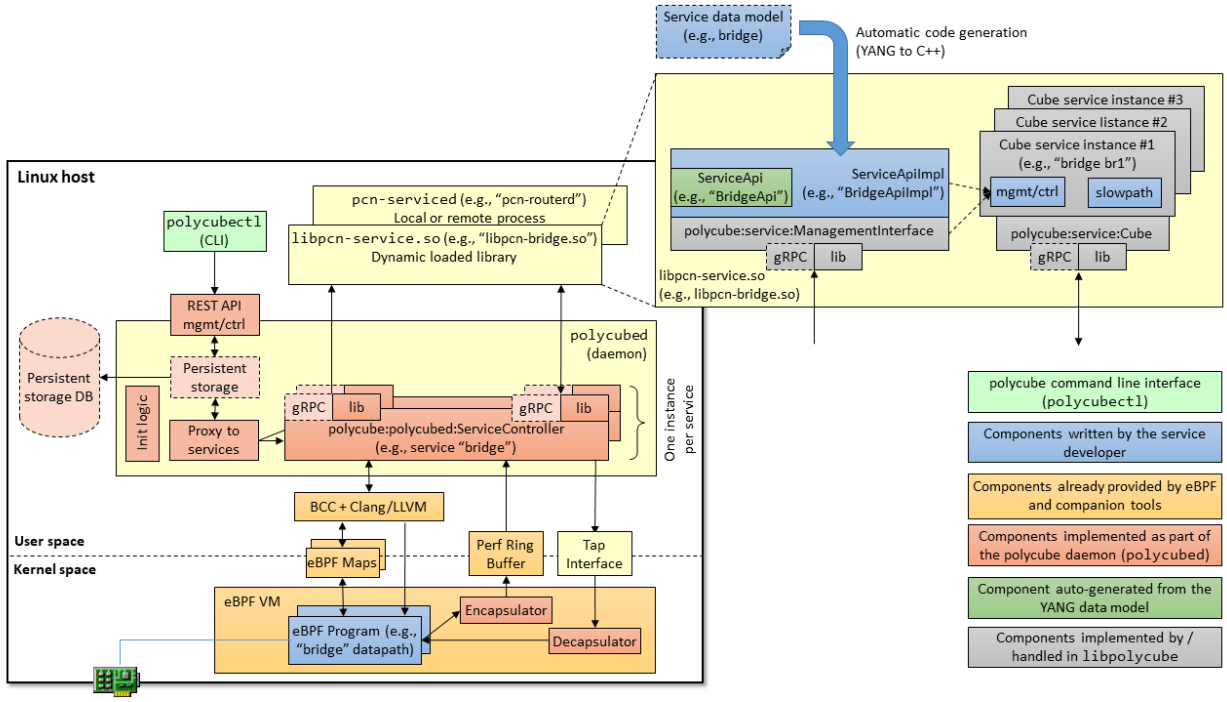
Hooray, you have Polycube installed and ready to be used, please refer to *Quick Start* to start using your installation.

CHAPTER

FOUR

POLYCUBE ARCHITECTURE

The Polycube architecture is depicted below:



The architecture includes four main components: *polycubed*: a service-agnostic daemon to control the polycube service, a set of *polycube services*, which actually implement the network functions, *polycubectl*, a service-agnostic CLI that allows to interact with polycube and all the available services, and *libpolycube*, a library that keeps some common code to be reused across multiple network functions.

4.1 The system daemon: polycubed

polycubed is the main component of the architecture. It is a service-agnostic daemon that allows to control the entire polycube service, such as starting, configuring and stopping all the available network functions (a.k.a., *services*). This module acts mainly as a proxy: it receives a request from its northbound REST interface, it forwards it to the proper service instance, and it returns back the answer to the user.

Polycube supports both **local** services, implemented as shared libraries, which are installed in the same server of *polycubed* and whose interaction is through direct calls, and **remote** services, implemented as remote daemons possibly running on a different machine, which communicate with *polycubed* through gRPC. Polycubed hides the above details and allow to get access to any registered service, either local or remote, in the same way from its REST interface. For

the same reason, a service will be able to interact to the *polycubed* daemon independently from the fact that the service itself is remote or local, hence facilitating the service developer who does not have to deal with the low-level details of the communication with the daemon.

4.2 The command line interface: polycubectl

Polycubectl implements the service-agnostic command-line interface (CLI) that enables to control the entire system, such as starting/stopping services, creating service instances (e.g., bridges *br1* and *br2*), and configuring / querying the internals of each single service (e.g., enabling the spanning tree on bridge *br1*).

As in case of *polycubed*, this module cannot know, a priori, which service will have to control; hence its internal architecture (not detailed here for the sake of brevity) is service-agnostic, meaning that is able to interact with any service through well-defined control/management interface that has to be implemented in each service. In order to facilitate the programmer in creating polycube services with the proper interface, we provide a set of automatic code generation tools that can create the skeleton of the control/management interface starting from the YANG data model of the service itself.

4.3 Polycube services

The flexible architecture of Polycube enables the support of multiple services (a.k.a., *network functions* such as bridge, router, NAT, etc.), even not known a-priori. Polycube services are similar to *plug-ins* that can be installed and launched at run-time. Obviously, each service has to implement a specific interface to be recognized and controlled by polycubed.

Each network function is implemented as a separated module and multiple flavors of the same function can coexist, such as in case of a simple (and fast) version to be used when speed is important and a reduced set of features can suffice, while a more complete (but slower) version can be used in other cases.

Each service implementation includes the *datapath*, namely the eBPF code to be injected in the kernel, the *control/management plane*, which defines the primitives that allow to configure the service behavior, and the *slow path*, which handles packets that cannot be fully processed in the kernel. While the former code runs in the Linux kernel, the latter components are executed in user-space. Given the feature-rich nature of the eBPF, the slow path should be rather small; an example can be the implementation of the spanning tree for 802.1Q bridges, which does not make sense to have in the kernel and that can stay in user space.

4.4 The common library for Polycube services: libpolycube

This library contains some common code that can be reused across multiple network functions. In facilitates the creation of links between services (i.e., to create a *service chain*), it provides common primitives such as a logging system, it facilitates the access to eBPF maps, and more. This library leverages [BCC](https://github.com/iovisor/bcc/) for compiling and injecting eBPF programs into the kernel.

CHAPTER

FIVE

CUBES ARCHITECTURE

A `Cube` is a logical entity, composed by a `data plane` (which can include one or more eBPF programs) and a `control/management plane`.

In addition, a Cube may include also a `slow path` in case some additional data plane-related code is required to overcome some limitations of the eBPF data plane. This is the case of flooding packets in an 802.1 bridge, when the MAC destination address does not exist (yet) in the filtering database. In fact, this action (which requires a packet to be sent on all active ports except the one on which it was received) involves redirecting a packet on multiple interfaces, an operation that is not supported by the eBPF technology. In addition, this operation may require a loop in the dataplane, i.e., cycling on the list of interfaces and send them the packet, which is hard to do due to the eBPF limitations.

5.1 Types of Cubes

Polycube has two diferent kind of cubes, **standard cubes** and **transparent cubes**.

Standard cubes have forwarding capabilities, such router and bridge.

A standard cube:

- defines a *ports* concept, ports are *connected* to other ports or netdevs;
- makes a **forwarding** decision, i.e, sending the packet to another port;
- it follows middle-box model, is a network function with multiple ports;

A transparent cube is a cube without any forwarding capability, such as a network monitor, NAT and a firewall.

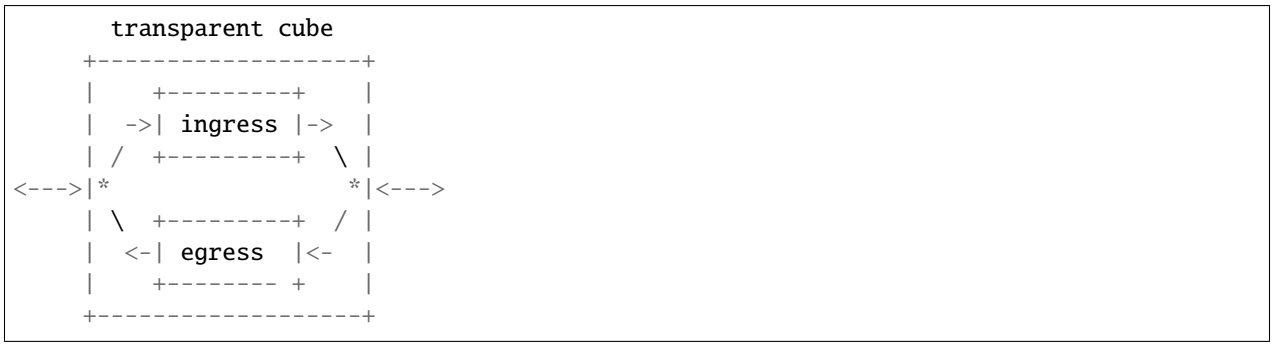
A transparent cube:

- does not define any port, hence it cannot be *connected* to other services, but *attached* to an existing port;
- it can be *attached* to an existing port such as the one of a normal service (e.g., port1 on router2) or a network device (netdev) in the host (e.g., *veth0* or *eth0*);

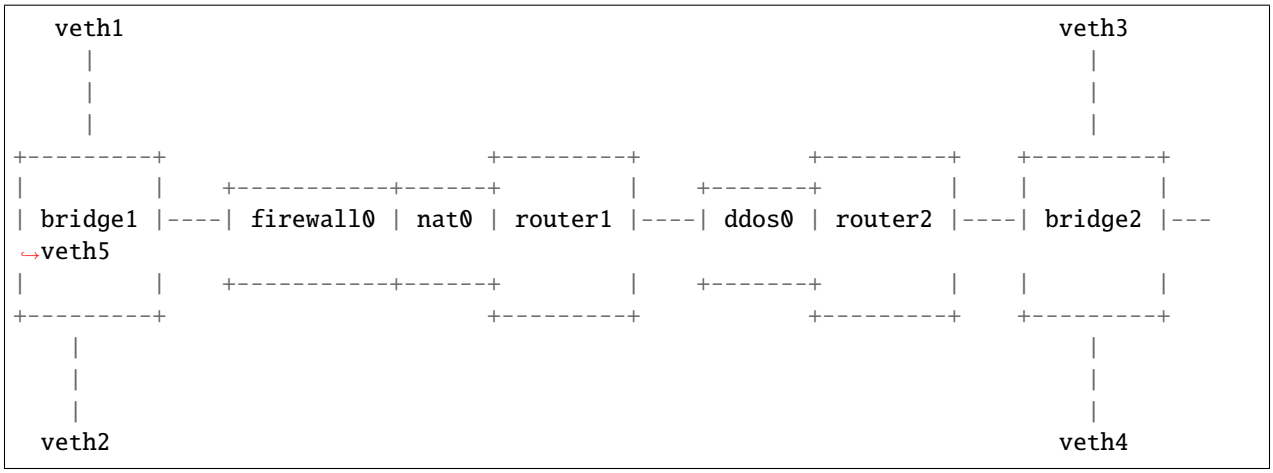
- it *inherits* all the parameters associated to the port it is attached to (e.g., MAC/IPv4 addresses, link speed, etc.);
- it allows to have a *stack* of transparent services on top of a port, very similar to a stack of functions.

A transparent cube can define two processing handlers, *ingress* and *egress*, which operate on two possible traffic directions:

- *ingress*: it handles the traffic that goes *toward* the port it is attached to. In case of a netdev (e.g., *eth0*), this selects the traffic that comes from the external world and goes toward the TCP/IP stack of the host. In case of a cube port, it is the traffic that is *entering* in the network function.
- *egress*: it handles the traffic that comes from the cube/netdev it is attached to. In case of a netdev (e.g., *eth0*), it selects the traffic that comes from the TCP/IP stack and *leaves* the host from that port. In case of a cube port, it is the traffic that is *leaving* the network function.



Following is example topology composed by standard and transparent cubes.



`polycubectl ?` shows available cubes installed on your system.

A shadow cube:

Only a standard cube can be Shadow type;

- `polycubectl <cubetype> add <cubename> shadow=true`.

A shadow cube is associated with a Linux network namespace;

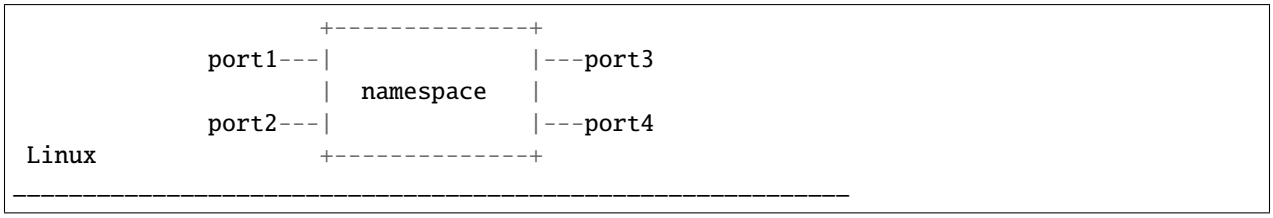
The parameters between the shadow cube and the namespace are aligned;

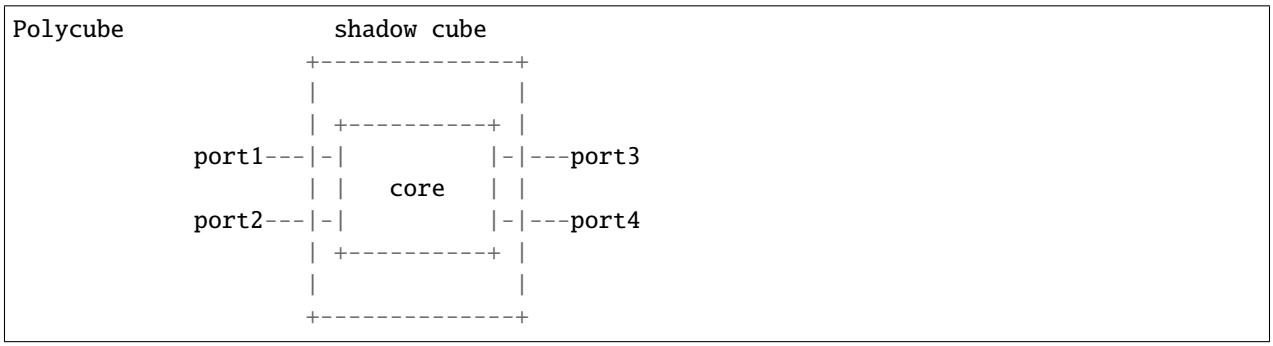
A port defined on a shadow cube is also visible from the network namespace:

- the user can decide to configure the ports using Linux (e.g. ifconfig or the ip command) or polycubectl;

for example: "`polycubectl <cubename> ports <PortName> set ip=<IpAddress>`" it is the same as "`ip netns exec pcn-<cubename> ifconfig <PortName> <IpAddress>`".

- the developer can let Linux handle some traffic by sending it to the namespace (e.g. ARP, ICMP, but in general all those protocols able to be managed by a tool running inside the namespace);





5.2 Cubes structure

5.2.1 Cubes Instances

Cubes are created by the `polycubectl <cubetype> add <cubename>` command, for example:

```
# create a router instance called r1
polycubectl router add r1
# create a simplebridge instance br1
polycubectl simplebridge add br1
```

5.2.2 Create Ports

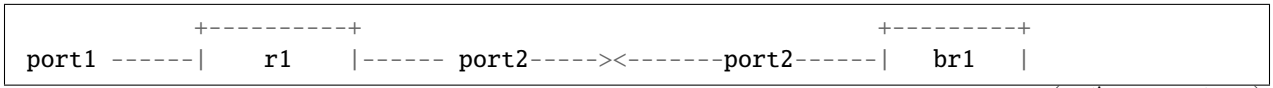
Cubes can send/receive traffic through `ports`.

NOTE: Just create a port does NOT allow to send/receive traffic.

Ports are created using `polycubectl <cubename> ports add <portname> [parameter=value, [parameter1=value1, ...]]`.

In order to `send/receive traffic`, the user has to setup the `peer` value or use the equivalent `connect` primitive. More details next.

Ports are logical entities and need to be connected to (physical/virtual) network interfaces or to other ports to be fully operational.



(continues on next page)

(continued from previous page)

```
(netdev)     |  (cube)  |    (cube port)        (cube port)    |  (cube) |
peer=veth1   +----------+  peer=br1:port2      peer=r1:port2  +---------+
                    |
                    |
                  port3
                 (netdev)
                peer=eth0
```

For instance:

```
# create port2 on br1 (simplebridge), it doesn't require any further parameters
polycubectl br1 ports add port2

# create portX on r1 (router), it doesn't require mandatory parameters, but it is useful␣
↪to assign an ip (during or after creation)
polycubectl r1 ports add port1 ip=10.0.1.1/24
polycubectl r1 ports add port2 ip=10.0.2.1/24
polycubectl r1 ports add port1 ip=10.0.3.1/24
```

5.2.3 Connect Ports

Two primitives are available: `set peer` or `connect`.

5.3 Set peer

The `peer` parameter defines where the port is connected to, it is possible to connect ports to linux netdevs or to ports belonging to other cubes.

- set peer to a `netdev` name in order to connect to it, (`eth0`, `wlan0`, `veth1`...)
- set peer to `cube_name:port_name` to connect the port to the port of another cube. (e.g. `br1:port1`). In this case the peer on both ports have to be set in order to create the connection.

If the `peer` is empty it means the port is down, so packets are not received from it, and packets sent through it are dropped.

Following is an example, referred to previous picture.

```
#using set peer
polycubectl r1 ports port1 set peer=veth1
polycubectl r1 ports port3 set peer=eth0
polycubectl r1  ports port2 set peer=br1:port2
polycubectl br1 ports port2 set peer=r1:port2
```

5.4 Connect

The `connect` primitive provides an alternative way to connect ports.

- connect to a `netdev` - Use `polycubectl connect <cube1>:<port1> <netdev>`
- connect to `cube_name:port_name` - Use `polycubectl connect <cube1>:<port1> <cube_name>:<port_name>`

Following is an example, referred to previous picture.

```
#using connect
polycubectl connect r1:port1 veth1
polycubectl connect r1:port3 eth0
polycubectl connect r1:port2 br1:port2
```

5.4.1 Attach and Detach primitives

These primitives allow to associate transparent cubes to standard cube's ports or to netdevs on the system.

```
polycubectl attach firewall1 r1:port2

polycubectl attach firewall0 veth1
```

5.5 Hook points

A Polycube service can be instantiated in a way that works when attached to either `TC`, `XDP_SKB` or `XDP_DRV` eBPF hooks. The `XDP_DRV`, which can guarantees the best performance, can be used only if it is supported by your NIC driver.

To create a service cube and select the desired hook point, just use this command:

```
polycubectl <service name> add <cube name> type=<hook point>
```

For example if you want to create an instance of the router service working with the XDP_SKB hook point:

```
polycubectl router add r1 type=XDP_SKB
```

By default, a service is instantiated in `TC` mode.

5.6 Traffic debugging with Span Mode

While the traffic flowing on a network device (e.g., veth0) can be captured and analyzed with tools such as Wireshark or tcpdump, the traffic flowing through an eBPF chain (e.g, a Polycube router connected to a Polycube bridge) is not visible outside eBPF, hence it cannot be captured by a sniffing program.

To overcome this problem, shadow cubes have a mode called **span** that allows to duplicate (hence, capture and analyze) all the traffic flowing through the ports of the cube to the corresponding virtual ethernet devices of the linked namespace. For those who come from 'traditional' hardware networking, _span_ is the term used when you want to duplicate the traffic from a first port of a network device to a second port of the same network device, e.g., to analyze the incoming

data. Span mode is very useful for debugging; a traditional sniffing program such as Wireshark or Tcpdump can sniff all the traffic flowing through a shadow cube, selecting each port.

To activate the span mode, use the following command: `polycubectl <cubename> set span=true`.

Note that the span mode consumes many resources when it is active, so it is disabled by default; we recommend to use it only when needed.

Note. Span mode duplicates traffic in the dedicated namespace, but the cube continues to handle traffic as usual. This could create some problems when the Linux kernel is involved in the processing. For example, if we have a shadow router with active span mode we should avoid to activate the IP forwarding on Linux, otherwise the router service forwards packets and copies them to the namespace, the namespace forwards again packets and there will be a lot of duplicated traffic.

CHAPTER

SIX

POLYCUBED: THE POLYCUBE SYSTEM DAEMON

The Polycube system daemon (polycubed) is in charge of managing the lifecycle of cubes, such as creating/updating/deleting network services.

In addition, it provides a single point of entry (a rest API server) for the configuration of any network function.

The preferred way to interact with polycubed is through *polycubectl*.

The Polycube System Daemon `polycubed` provides a kernel abstraction layer that is used by the different services. It exposes a configuration mechanism of the different service instances through a rest API server. Users can interact with it using *polycubectl*.

It requires root privileges and only an instance can be launched system wide.

6.1 Systemd integration

`polycubed` can be managed as a systemd service.

```
# start the service
sudo systemctl start polycubed

# stop the service
sudo systemctl stop polycubed

# restart the service
sudo systemctl reload-or-restart polycubed

# enable the service to be started at boot time
sudo systemctl enable polycubed

# see service status
sudo systemctl status polycubed

# check logs
journalctl -u polycubed // '-f' can be used to see a live version
```

6.2 Usage

```
Usage: polycubed [OPTIONS]
-p, --port PORT: port where the rest server listens (default: 9000)
-a, --addr: addr where polycubed listens (default: localhost)
-l, --loglevel: set log level (trace, debug, info, warn, err, critical, off, default:␣
↪info)
-c, --cert: path to ssl certificate
-k, --key: path to ssl private key
--cacert: path to certification authority certificate used to validate clients
-d, --daemon: run as daemon (default: false)
-v, --version: show version and exit
--logfile: file to save polycube logs (default: /var/log/polycube/polycubed.log)
--pidfile: file to save polycubed pid (default: /var/run/polycube.pid)
--configfile: configuration file (default: /etc/polycube/polycube.conf)
--cert-black-list: path to black listed certificates
--cert-white-list: path to white listed certificates
-h, --help: print this message
```

6.3 Configuration file

The configuration file is and alternative way to configure polycubed. It is a text file using a `parameter: value` syntax, where parameter refers to the long ones described above. By default it resides in `/etc/polycube/polycubed.conf`, this location can be changed by using the `--configfile` parameter. A default file will be created if it does not exist and the `--configfile` parameter is not specified. If the same parameter is specified in both, the configuration file and the command line, the later one is considered. `polycubed` has to be restarted after the configuration file is updated in order to take the new values.

```
# polycubed configuration file
# this is a comment
loglevel: info
daemon: true
#p: 6000 <-- this is NOT supported, only long options are
```

6.4 Persistency

Polycubed has persistent capabilities which means that (1) it can automatically load the configuration that was present when the daemon was shut down, (2) each time a configuration command is issued, it is automatically dumped on disk. This allows polycubed also to recover from failures, such as rebooting the machine. To enable this feature we need to start polycubed with the `--cubes-dump-enable` flag. The daemon keeps in memory an instance of all the topology, including the configuration of each individual service. Topology and configuration are automatically updated at each new command; the configuration is also dumped to disk, on file `/etc/polycube/cubes.yaml`. The standard behavior of the daemon at startup is to load, if present, the latest topology that was active at the end of the previous execution. Users can load a different topology file by using the `--cubes-dump-file` flag followed by the path to the file. In case we want to start polycubed with an empty topology, avoiding any possible load at startup, we can launch polycubed with the `--cubes-dump-clean-init` flag. Beware that in this case any existing configuration in the default file will be overwritten. `--cubes-dump-enable` is required if we want to use any of the other two related flags. There are some limitations: (1) YANG actions, such as "append" for firewall and nat rules, are not supported, (2) some services

fail to load the full configuration at once and (3) transparent services attached to netdevs are not saved in the cubes dump file.

```
# start daemon with topology saving functionalities
polycubed --cubes-dump-enable

# start polycubed with custom cubes configuration
polycubed --cubes-dump-enable --cubes-dump-file ~/Desktop/myCubes.yaml

# start polycubed with an empty topology
polycubed --cubes-dump-enable --cubes-dump-clean-init
```

6.5 Debugging

The debugging of polycubed can be turned on by starting the daemon with the `--loglevel=debug` flag.

6.6 Rest API

Here is the detailed documentation of the REST API.

CHAPTER

SEVEN

POLYCUBECTL: THE COMMAND-LINE INTERFACE FOR POLYCUBE

`polycubectl` is the Command Line Interface (CLI) for Polycube.

`polycubectl` is a generic CLI, that enables the user to interact with `Cubes` (`bridge`, `router`, ...) and with some framework primitives to `connect`, `show` and build complex `topologies`.

`polycubectl` does not need to be modified when a new cube is developed ad added to Polycube. Its service-agnostic nature, thanks to the use of YANG data models, enables the CLI to be service-independent.

7.1 Install

`polycubectl` is installed by default with `Polycube`. Refer to *quickstart* or general *install* guide for more info.

7.2 How to use

NOTE: `polycubed` must be running in order to use `polycubectl`. You can start the daemon typing `sudo polycubed` in another terminal. Refer to *Quick Start*.

```
# Show help
polycubectl --help
Keyword           Type      Description
simpleforwarder   service   Simple Forwarder Base Service
simplebridge      service   Simple L2 Bridge Service
router            service   Router Service
iptables          service   Iptables-clone Service
helloworld        service   Helloworld Service
ddosmitigator     service   DDoS Mitigator Service
firewall          service   Firewall Service
k8switch          service   Kubernetes HyperSwitch Service
k8sfilter         service   Kubernetes Traffic Filtering Service
lbdsr             service   LoadBalancer Direct Server Return Service
lbrp              service   LoadBalancer Reverse-Proxy Service
pbforwarder       service   Policy-Based Forwarder Service
bridge            service   Bridge Service
nat               service   NAT Service
packetcapture     service   Packetcapture Service

connect           command   Connect ports
disconnect        command   Disconnect ports
```

(continues on next page)

(continued from previous page)

```
attach            command   Attach transparent cubes
detach            command   Detach transparent cubes

services          command   Show/Add/Del services (e.g. Bridge, Router, ..)
cubes             command   Show running service instances (e.g. br1, nat2, ..)
topology          command   Show topology of service instances
netdevs           command   Show net devices available
```

The general syntax for `polycubectl` is the following:

```
polycubectl [parent] [command] [child] [argument0=value0] [argument1=value1]
```

- `parent`: path of the parent resource where the command has to be applied.
- `command`: `add`, `del`, `show`, `set` or a yang action.
- `child`: specific resource where the command is applied.
- `argument`: some commands accept additional commands that are sent in the body request.

Some examples:

```
polycubectl router r0 add loglevel=debug
polycubectl r0 ports add port1 ip=10.1.0.1 netmask=255.255.0.0
polycubectl r0 show routing table
polycubectl r0 ports port1 set peer=veth1
polycubectl r0 ports del port1

# yang action
polycubectl firewall1 chain ingress append src=10.0.0.1 action=DROP
```

The best way to know what is the syntax for each service is to use the *Help* or the bash completion by pressing <TAB> at any point.

`polycubectl` is also able to read the contents of the request from the standard input, it can be used in two ways:

7.2.1 Passing complex configuration from the command line

```
# create a helloworld instance with loglevel debug and action forward
polycubectl helloworld add hw0 << EOF
{
"loglevel": "debug",
"action": "forward"
}
EOF
```

7.2.2 Reading configuration from a file

```
# create helloworld from a yaml file
polycubectl helloworld add hw0 < hw0.yaml

# create a router from a json file
polycubectl router add r0 < r0.json

# add a list of cubes
polycubectl cubes add < mycubes.yaml
```

7.2.3 Help

`polycubectl` provides an interactive way to navigate help.

At each depth level the user can type `?` to get contextual help information.

The output of help command is basically a list of `keywords` that could be used (instead of `?`), and is some case a list of `parameters`.

Following is an example of a possible interaction with help, to configure a `router`.

```
polycubectl router ?

Keyword  Type      Description
add      command   Add entry to a list
del      command   Delete entry of a list
show     command   Show entry or list [-normal | -brief | -verbose | -json | -yaml]
<name>   string    Name of the router service
```

```
polycubectl router add ?

Keyword             Type     Description
<name>              string   Name of the router service

Other parameters:
loglevel=value      string   Logging level of a cube, from none (OFF) to the most␣
↪verbose (TRACE)

Example:
polycubectl router add router1 loglevel=INFO
```

```
polycubectl router add r1
```

7.2.4 Flags

The `show` command supports the `-normal`, `-brief`, `-verbose`, `-json`, `-yaml` flags that affects how the output is printed on the terminal.

-hide

The `-hide=arg0[arg1[,arg2...]]` flag allows to hide some elements in the output. It expects a list paths to resources to be hidden.

Examples:

```
# hide ports from output
polycubectl router r0 show -hide=ports

# hide uuid of ports
polycubectl router r0 show -hide=ports.uuid

# hide uuids and mac ports
polycubectl router r0 show -hide=ports.uuid,ports.mac
```

7.2.5 Tutorials

More complete examples are available in *tutorials*.

7.3 Configuration

By default, `polycubectl` contacts `polycubed` at `http://localhost:9000/polycube/v1/`. The user can override this configuration with following instructions.

7.3.1 Parameters

- `debug` shows HTTP requests and response sent and received by CLI
- `expert` enables the possibility to add new services at runtime
- `url` is the URL to contact polycubed
- `cert` client certificate when using https
- `key` client private key
- `cacert` certification authority certificate that signed the server's certificate

7.3.2 Configuration file

Configuration file is placed at `$HOME/.config/polycube/polycubectl_config.yaml`.

```
# debug: shows http method/url and body of the response
# expert: enables the possibility to add new services
# url: is the base URL to contact the rest server

debug: false
expert: true
url: http://localhost:9000/polycube/v1/
key: ""
cacert: ""
cert: ""
```

7.3.3 Environment variables

Following are available ENV variables:

```
POLYCUBECTL_DEBUG
POLYCUBECTL_URL
POLYCUBECTL_EXPERT
POLYCUBECTL_CERT
POLYCUBECTL_KEY
POLYCUBECTL_CACERT
```

A possible example of configuration is:

```
$ export POLYCUBECTL_URL="http://10.0.0.1:9000/polycube/v1/"
```

CHAPTER

EIGHT

SECURING THE POLYCUBED DAEMON

Polycube uses a security model based on X509 certificates to secure polycube daemon to polycube cli communication.

8.1 polycubed

8.1.1 Server authentication

In order to authenticate the server the `cert` and `key` parameters are needed.

Example:

```
# polycubed configuration file
cert: path to server certificate
key: path to server key
```

8.1.2 Client authentication

Polycubed supports thee different modes to perform client authentication.

Mode 1: Certification authority based

This method requires you to create own signed certificates for the clients. Please see *Create own certification authority with openssl* to generate the client certificates.

In this mode the following parameters are needed:

- `cert`: server certificate
- `key`: server key
- `cacert`: certification authority certificate used to authenticate clients

Any client with a certificate signed by that certification authority is able to use polycubed.

Configuration example:

```
# polycubed configuration file
# server certificate
cert: /home/user/server.crt
# server private key
key: /home/user/server.key
```

(continues on next page)

(continued from previous page)

```
# CA certificate used to authenticate clients
cacert: /home/user/ca.crt
```

Mode 2: Certification Authority based with blacklist

This mode is an extension to Mode 1 that allows to deny the access to given certificates by passing the `cert-black-list`, it is a folder containing hashed names for the banned certificates. See *How to generate hash links to certificates*.

Configuration example:

```
# polycubed configuration file
cert: /home/user/server.crt
# server private key
key: /home/user/server.key
# CA certificate used to authenticate clients
cacert: /home/user/ca.crt
# folder with blacklisted certificates
cert-black-list: /home/user/my_black_list/
```

Mode 3: Whitelist based

This mode allows to use already existing client certificates by providing the `cert-white-list` parameter that is a folder containing hash named client certificates allowed to access polycubed. See *How to generate hash links to certificates*.

Configuration example:

```
# polycubed configuration file
cert: /home/user/server.crt
# server private key
key: /home/user/server.key
# folder with allowed certificates
cert-white-list: /home/user/my_white_list/
```

8.2 polycubectl

To enable a secure connection to polycubed the user has configure the following parameters for polycubectl. See *polycubectl configuration* to get more details.

- `url`: must start with `https`
- `cert`: client certificate
- `key`: client private key
- `cacert`: certification authority certificate that signed the server certificate

8.2.1 Create own certification authority with openssl

You can create your own certification authority to issue certificates for polycubed and the clients.

1. Create certification authority

   a. Create root key

```
openssl genrsa -des3 -out ca.key 4096
```

   Remove the `-des3` if you don't have to protect the key wiht a passphrase

   b. Create root certificate

```
openssl req -x509 -new -nodes -key ca.key -sha256 -days 1024 -out ca.crt
```

2. Create polycubed certificate

   This step can be skipped if you already have a valid certificate to be used.

   a. Create polycubed private key

```
openssl genrsa -out server.key 2048
```

   b. Create certificate request for polycubed

```
openssl req -new -key server.key -out server.csr
```

   c. Generate server certificate

   The server certificate must have the alternative name set to the IP or domain where polycubed will run

   Create a server.conf file with the following content. Add the DNS entries you need.

```
[req_ext]
subjectAltName = @alt_names
[alt_names]
DNS.1 = localhost
DNS.2 = 127.0.0.1
```

   Create certificate

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -out server.crt -days 1024 -sha256 -extfile server.conf -extensions req_
↪ext
```

3. Create client certificate

   a. Create client key

```
openssl genrsa -out client.key 2048
```

   b. Create client certificate request

```
openssl req -new -key client.key -out client.csr
```

   c. Generate client certificate

```
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
    -out client.crt -days 1024 -sha256
```

Please keep a copy of the client certificates you generate, they could be uselful in the future in case you want to use the `cert-black-list` option.

8.2.2 How to generate hash links to certificates

The `cert-black-list` and `cert-white-list` parameters refer to a folder that contains certificates named by their hash value.

Follow these instructions to generate hash links to certificates:

```
# copy the certificates to your black or whitelist folder
$ cp client.crt myfolder/
$ cd myfolder
$ ls
client.crt
# create symbolic links
$ c_rehash .
Doing .
$ ls -l
9d75b5b3.0 -> client.crt
client1.crt
eb7bf4cd.0 -> client.crt
```

Please see the c_rehash tool to get more information.

CHAPTER

NINE

PCN-IPTABLES: A CLONE OF IPTABLES BASED ON EBPF

Polycube includes the `pcn-iptables` standalone application, a stateful firewall whose syntax and semantic are compatible with the well-known `iptables` Linux tool.

The frontend provides the same CLI of iptables; users can set up security policies using the same syntax by simply executing `pcn-iptables` instead of `iptables`. The backend is based on *eBPF* programs, more efficient classificaiton algorithms and runtime optimizations; the backend runs as a dedicated service in Polycube.

9.1 Supported features

Currently supported features:

- Support for `INPUT`, `OUTPUT`, `FORWARD` chains
- Support for `ip`, `protocol`, `ports`, `tcp flags`, `interfaces`
- Support for `connection tracking`
- Support for bpf `TC` and `XDP` mode

Detailed supported parameters

- `-s` source IP
- `-d` destination IP
- `-p` l4 protocol
- `--sport` source port
- `--dport` destination port
- `--tcpflags` tcp flags
- `-i` input interface
- `-o` output interface
- `-m conntrack --ctstate` conntrack module

Detailed supported targets

- `-j ACCEPT` accept traffic
- `-j DROP` drop traffic

Detailed supported commands

- `-S` Show rules
- `-L` List rules

- `-A` Append rule
- `-I` Insert rule
- `-D` Delete rule
- `-P <CHAIN> DROP/ACCEPT` Setup default policy for `<CHAIN>`
- `-F <CHAIN>` Flush policies for `<CHAIN>`

9.1.1 Limitations

- No support for multiple chains
- No support for `SNAT`, `DNAT`, `MASQUESRADE`
- `-S -L` generate an output slightly different from iptables

9.2 Install

9.2.1 Prerequisites

pcn-iptables comes as a component of polycube framework. Refer to *polycube install guide* for dependencies, kernel requirements and basic checkout and install guide.

9.2.2 Install

To compile and install `pcn-iptables`, you should enable the `ENABLE_PCN_IPTABLES` flag in the polycube CMakeFile, which is set to `OFF` by default; this allows to compile the customized version of `iptables` used to translate commands, and install in the system pcn-iptables-init pcn-iptables and pcn-iptables-clean utils.

Note: The `ENABLE_SERVICE_IPTABLES` flag, which is set to `ON` by default, is used to compile and install the `libpcn-iptables.so` service (like other polycube services: bridge, router, ..). This flag is required to be enabled as well, but it comes by default.

```
cd polycube

# Note: ensure git submodules are updated
# git submodule update --init --recursive

mkdir -p build
cd build
cmake .. -DENABLE_PCN_IPTABLES=ON
make -j`nproc` && sudo make install
```

9.3 Run

9.3.1 1. Initialize pcn-iptables

```
# Start polycubed, in other terminal (or background)
sudo polycubed --daemon
# Initialize pcn-iptables
pcn-iptables-init
```

9.3.2 2. Use pcn-iptables

`pcn-iptables` provides same iptables syntax. Please refer to `iptables` online docs for more info. Following are just few examples of available commands.

```
# E.g.
pcn-iptables -A INPUT -s 10.0.0.1 -j DROP # Append rule to INPUT chain
pcn-iptables -D INPUT -s 10.0.0.1 -j DROP # Delete rule from INPUT chain
pcn-iptables -I INPUT -s 10.0.0.2 -j DROP # Insert rule into INPUT chain

# Example of a complex rule
pcn-iptables -A INPUT -s 10.0.0.0/8 -d 10.0.0.2 -p tcp --sport 9090 --dport 80 --
↪tcpflags SYN,ACK ACK -j DROP

# Example of a conntrack rule
pcn-iptables -A OUTPUT -m conntrack --ctstate=ESTABLISHED -j ACCEPT

# Show rules
pcn-iptables -S # dump rules
pcn-iptables -L INPUT # dump rules for INPUT chain

pcn-iptables -P FORWARD DROP # set default policy for FORWARD chain
```

NOTE: do *not* use use `sudo pcn-iptables ...`

9.3.3 3. Stop pcn-iptables

```
# Stop and clean pcn-iptables
pcn-iptables-clean

# Execute the below command to validate if cleanup is successful.
pcn-iptables -S
``Note:  On successful cleanup, you should receive "No cube found named pcn-iptables"``
```

9.4 Advanced Features

9.4.1 XDP mode

`pcn-iptables` can also be run in `XDP` mode. This mode comes with performance gain, especially when policy are configured to DROP traffic.

```
pcn-iptables-init-xdp
```

9.4.2 Limitations

- pcn-iptables operates only on interfaces that support XDP native mode
- traffic is not filtered on interfaces that support only eBPF TC programs.

9.5 pcn-iptables components

`pcn-iptables` is composed by three main components:

1. `pcn-iptables` service (`src/services/pcn-iptables`): a Polycube service, a special one since performs some extra work, but basically expose its API and CLI, according to Polycube standard.

9.5.1 iptables submodule

A customized fork of iptables is included as submodule under src/components/iptables/iptables. This modified version of iptables is in charge of validate commands, translate them from iptables to polycube syntax, then forward them to pcn-iptables service instead of pushing them into the kernel via netfilter.

9.5.2 scripts folder

Scripts are used as a glue logic to make pcn-iptables run. Main purpose is initialize, cleanup and run pcn-iptables, pass pcn-iptables parameters through iptables (in charge of converting them), then pass converted commands to pcn-iptables service. Scripts are installed under `/usr/local/bin`.

9.6 pcn-iptables components

`pcn-iptables` is composed by three main components:

1. `pcn-iptables` service (`src/services/pcn-iptables`): a Polycube service that is especially tailored to work with the `pcn-iptables` executable; as usual, it exposes its API and CLI according to Polycube standard.
2. `iptables*` (`src/components/iptables/iptables`): a modified version of iptables, in charge of validate commands, translate them from iptables to polycube syntax, then forward them to pcn-iptables service instead of push them into the kernel via netfilter.
3. `scripts` (`src/components/iptables/scripts`): this is a folder containing some glue logic and scripts to initialize, cleanup and use `pcn-iptables`. `pcn-iptables` itself is a script that forwards commands to `iptables*` (2), then forwards the translated command to `pcn-iptables` (1). Scripts are installed under `/usr/local/bin`.

CHAPTER

TEN

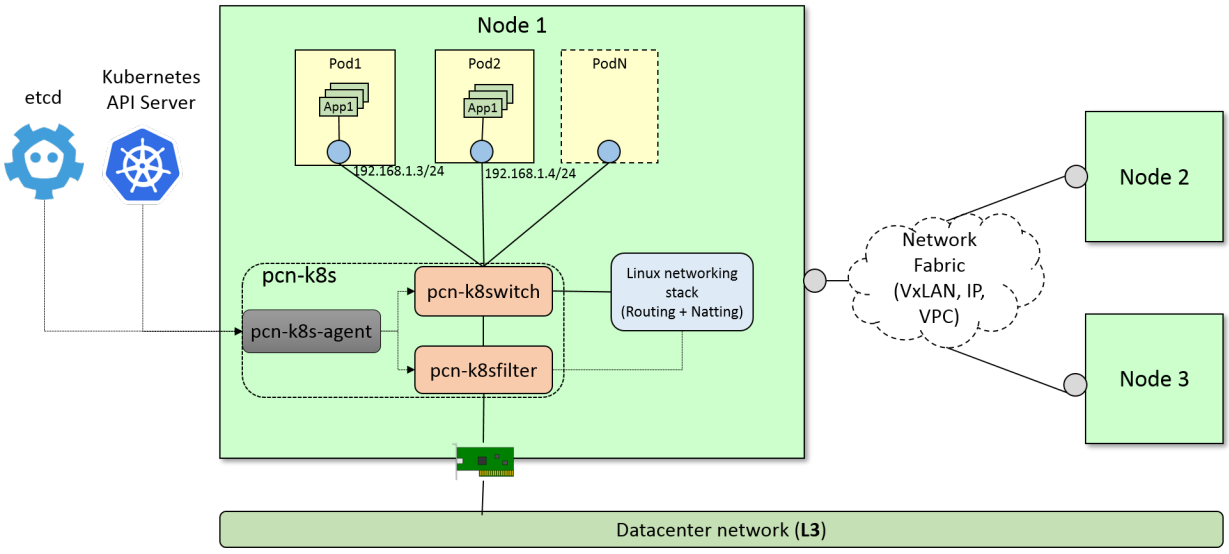
PCN-K8S: A NETWORK PROVIDER FOR KUBERNETES

10.1 Introduction

`pcn-k8s` leverages some polycube services to provide network support for pods running in Kubernetes. It supports the cluster Kubernetes networking model, `ClusterIP` and `NodePort` services. Security policies and `LoadBalancing` service mode are not yet supported.

`pcn-k8s` is made of 3 components:

- The `pcn-k8s-agent`, which interfaces with the Kubernetes API master service and dynamically reconfigures the networking components in order to create the network environment required by Kubernetes and provide connectivity to the pods.
- The `pcn-cni` plugin implements CNI specification, it connects new pods to the `pcn-k8s` networking.
- `pcn-k8switch` and `pcn-k8sfilter` two polycube services that implement an eBPF datapath for kubernetes pod networking.



10.1.1 Networking Mode

`pcn-k8s` supports different methods to communicate pods running in different hosts

- **overlay networking**: when nodes are on different subnets and the user does not have direct control over the physical network an overlay networking is used. The default (and only supported yet) technology is `VxLAN`
- **direct routing**: when nodes are on the same subnet packets can be exchanged between nodes without encapsulating them
- **vpc**: when nodes run on a cloud provider that supports Virtual Private Cloud (VPC). As of today only *aws* is supported.

See *Configuring pcn-k8s* to get more info about how to configure the different modes.

10.1.2 Compatibility

pcn-k8s` is compatible with all versions of Kubernetes equal or greater than 1.9, although we recommend the latest version.

10.2 Installation

You may choose either of the below options.

1. Quick setup with `vagrant` (development environment)
2. Using `kubeadm` on Bare-Metal or VMs (Single or HA cluster)

10.2.1 1. Quick Setup with vagrant

- The fastest mode to test `pcn-k8s` including setup.

Prerequisites

Download and set up the following packages.

- Vagrant (Tested on 2.2.4)
- VirtualBox (Tested on 6.0.4)

Follow the instruction below, after the pre-requisite:

1. Use this PCN-K8S Vagrantfile for your setup.
2. Execute `vagrant up` to bring all the nodes up and running.
3. `vagrant status` to check all the nodes and it's status
4. `vagrant ssh <node-name>` to SSH to the node.

Note: This vagrant setup takes care of setting up the kubeadm and joining the nodes along with the `pcn-k8s` CNI.

10.2.2 2. Using kubeadm on Bare-Metal or VMs (Single or HA cluster)

The easiest way to get started with `pcn-k8s` using `kubeadm`.

Please follow the kubeadm installation guide for the most up-to-date instructions.

Once `kubeadm` is installed, you can create the cluster. The following commands are intended to be used as a quick guide; please refer to the official single cluster setup guide or HA cluster setup guide for more detailed information.

Initialize master

First, you have to initialize the master by specifying the set of network addresses to be used for pods, such as the following:

```
sudo kubeadm init --pod-network-cidr=192.168.0.0/16

# In case you're using a multi-cloud cluster or your master has a
# PUBLIC IP/FQDN you may use the below command instead:
# (Replace the --apiserver-cert-extra-sans value with External IP or FQDN)
sudo kubeadm init --pod-network-cidr=192.168.0.0/16 \
    --apiserver-cert-extra-sans="Master External IP or FQDN"
```

This step will print on screen a command like:

```
kubeadm join --token <token> <master-ip>:<master-port> \
    --discovery-token-ca-cert-hash sha256:<hash>
```

Please save the `kubeadm join ...` command that is printed on screen, which will be used to join the workers nodes later.

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Verify that the system pods are in running state before continuing:

```
kubectl get pods --all-namespaces
```

You should see something like:

```
NAMESPACE     NAME                                 READY     STATUS    RESTARTS   AGE
kube-system   etcd-k8s-master                      1/1       Running   0          1m
kube-system   kube-apiserver-k8s-master            1/1       Running   0          1m
kube-system   kube-controller-manager-k8s-master   1/1       Running   0          1m
kube-system   kube-dns-545bc4bfd4-czf84            0/3       Pending   0          1m
kube-system   kube-proxy-hm4ck                     1/1       Running   0          1m
kube-system   kube-scheduler-k8s-master            1/1       Running   0          1m
```

`pcn-k8s` requires an `etcd` deployment to work, set the `etcd_url` parameter in pcn-k8s.yaml. standalone_etcd.yaml provides a basic etcd service that can be used for testing, deploy it before deploying pcn-k8s.

At this point you can install `pcn-k8s`:

```
kubectl apply -f https://raw.githubusercontent.com/polycube-network/polycube/master/src/
↪components/k8s/pcn-k8s.yaml
# It will take some time until the images are pulled.
```

```
# Optional: if you want to execute pods on the master node
kubectl taint nodes --all node-role.kubernetes.io/master-
```

Add workers

Workers can be added by executing the previously saved `kubeadm join` command on each new node, as shown in this example (please note that the actual command will be different on your system):

```
sudo kubeadm join --token 85856d.feb1e886dd94f7d5 130.192.225.143:6443 \
    --discovery-token-ca-cert-hash␣
↪sha256:2c3f07b126bdc772e113306f1082ece6c406f130704a1e08a9c67c65542b869d
```

You can see all the nodes in the cluster using the following command:

```
kubectl get nodes -o wide
```

After that, the cluster will be ready to accept requests and deploy pods.

Removing pcn-k8s

In order to remove `pcn-k8s` execute on the master node:

```
kubectl delete -f https://raw.githubusercontent.com/polycube-network/polycube/master/src/
↪components/k8s/pcn-k8s.yaml
```

10.3 Configuring pcn-k8s

`pcn-k8s` uses `etcd` to save the different configuration parameters. It is exposed at port *30901* of the master node if you used the *standalone_etcd.yaml* template to deploy it.

10.3.1 Installing etcdctl

The easiest way to get `etcdctl` is to download a etcd release and take the binary from there.

The different per-node parameters that the user can configure are:

- **directRouting (boolean)**: when this is enabled `pcn-k8s` will avoid to create tunnels among adjacent nodes (nodes that are on the same subnet).

example:

```
ETCDCTL_API=3 etcdctl --endpoints=130.192.225.145:30901 \
    put /nodes/node1/directRouting true
```

Note that in order to use that feature *directRouting* must be enabled in both nodes.

- **vpcMode**: specifies the kind of Virtual Provide Cloud where the node is running in. When this is set to a value provider (only *aws* is supported now) it configures the VPC and avoid creating tunnels to other nodes running on the same VPC. If this is empty, the vpc support is disabled.

```
ETCDCTL_API=3 etcdctl --endpoints=130.192.225.145:30901 \
    put /nodes/node1/vpcMode aws
```

- **publicIP**: In deployments where nodes are behind a NAT, you need to manually configure the public IP of the nodes in order to allow `pcn-k8s` to reach them from the external world. A typical example is when nodes are installed in different cloud providers, such as Amazon and Google, but are all part of the same k8s instance. In this case you can use the following command, which has to be repeated for each node that is behind the NAT:

```
ETCDCTL_API=3 etcdctl --endpoints=130.192.225.145:30901 \
    put /nodes/node1/publicIP 198.51.100.100
```

10.3.2 Running in *aws*

In order to let `pcn-k8s` interact with *aws* an *Identity and Access Management (IAM)* role is needed.

1. Create Policy: Go to the *IAM* Management Console, then select *Policies* on the left and then *Create policy* with the following JSON content:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "ec2:DescribeInstances",
        "ec2:CreateRoute",
        "ec2:DeleteRoute",
        "ec2:ModifyInstanceAttribute",
        "ec2:DescribeRouteTables",
        "ec2:ReplaceRoute",
        "iam:PassRole"
      ],
      "Resource": "*"
    }
  ]
}
```

2. Create Role: Go to the *IAM* Management Console, then select *Roles* on the left and then *Create role*. Select the *EC2* service, click on *Next: Permissions* button on bottom right, select the name of policy that you've created in the above step and click *Next: Review* button; set a name to the policy with some description and click on *Create role* button.
3. Attach Policy to role: Go to the *IAM* Management Console, then select *Roles* on the left and click on the *role name* that you've created in above step, go to *Permissions* tab and click *Attach policies*. Then search and select for *AmazonEC2FullAccess*, *IAMReadOnlyAccess* and *PowerUserAccess* policy and click on *Attach policy* to complete the step.

Assign the IAM role (that you've created in above step) to the EC2 instances while you create them.

Note: VxLAN exchanges traffic on port *4789/UDP*, be sure that you have configured security rules to allow it.

10.4 Testing your pcn-k8s installation

We present here some commands to test that your `pcn-k8s` deployment works as expected.

In order to run these tests, a cluster having at least two schedulable nodes (not tainted) is needed.

10.4.1 Deploy services and pods to test

```
kubectl create -f https://raw.githubusercontent.com/polycube-network/polycube/master/src/
↪components/k8s/examples/echoserver_nodeport.yaml
kubectl run curl1 --image=tutum/curl --replicas=5 --command -- sleep 600000
```

After a short period of time, all pods should be in the *Running* state

```
k8s@k8s-master:~$ kubectl get pods -o wide
NAME                            READY     STATUS    RESTARTS   AGE       IP              ␣
↪NODE
curl1-5df485555f-2dpwx          1/1       Running   0          1m        192.168.1.3     ␣
↪k8s-worker2
curl1-5df485555f-l7ql4          1/1       Running   0          1m        192.168.0.246   ␣
↪k8s-master
curl1-5df485555f-s9wsv          1/1       Running   0          1m        192.168.1.5     ␣
↪k8s-worker2
curl1-5df485555f-xh4wv          1/1       Running   0          1m        192.168.1.4     ␣
↪k8s-worker2
curl1-5df485555f-xqmxn          1/1       Running   0          1m        192.168.0.245   ␣
↪k8s-master
myechoserver-86856fd86f-fkzg6   1/1       Running   0          32s       192.168.1.6     ␣
↪k8s-worker2
```

10.4.2 Tests

The following section present some test cases to check everything is working as expected.

Test Node to Pod connectivity

```
# ping pod in master node
k8s@k8s-master:~$ ping 192.168.0.245

k8s@k8s-master:~$ ping 192.168.1.3
```

Test Pod to Pod connectivity

```
# select one pod running on master, in this case (192.168.0.245)
k8s@k8s-master:~$ ID=curl1-5df485555f-xqmxn

# ping pod in master
k8s@k8s-master:~$ kubectl exec $ID ping 192.168.0.246

# ping pod in worker
k8s@k8s-master:~$ kubectl exec $ID ping 192.168.1.5
```

Test Pod to Internet connectivity

```
# ping to internet
k8s@k8s-master:~$ kubectl exec $ID ping 8.8.8.8
```

Test ClusterIP service

The following command will give us the details about the service we created:

```
k8s@k8s-master:~$ kubectl describe service myechoserver
Name:                     myechoserver
Namespace:                default
Labels:                   app=myechoserver
Annotations:              <none>
Selector:                 app=myechoserver
Type:                     NodePort
IP:                       10.96.23.23
Port:                     <unset>  8080/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  31333/TCP
Endpoints:                192.168.1.6:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

```
# direct access to the backend
k8s@k8s-master:~$ curl 192.168.1.6:8080

# access from node to ClusterIP
curl 10.96.23.23:8080

# access from a pod (change ID to both, a pod in the local node and also a pod in a␣
↪remote node)
kubectl exec $ID curl 10.96.23.23:8080
```

Test NodePort service

The service is exposed in port *31333*, perform a request to the public IP of the master and the node from a remote host.

```
# request to master
curl 130.192.225.143:31333

# request to worker
curl 130.192.225.144:31333
```

TODO: - test dns service - test scale-up scale down

10.5 Troubleshooting

10.5.1 Recovering from a pcn-k8s failure

`pcn-k8s` expects a clean environment to start with and it is likely to fail if this is not verified. In case you hit any problems, please follow the next steps to recover from a failure:

```
#1. Remove pcn-k8s
kubectl delete -f https://raw.githubusercontent.com/polycube-network/polycube/master/src/
↪components/k8s/pcn-k8s.yaml

#2. Disable DNS
kubectl -n kube-system scale --replicas=0 deployment/kube-dns

#3. Remove garbage network interfaces and iptables(if any)
ip link del dev pcn_k8s
ip link del dev pcn_vxlan

#4. Relaunch pcn-k8s
kubectl apply -f https://raw.githubusercontent.com/polycube-network/polycube/master/src/
↪components/k8s/pcn-k8s.yaml

# Wait until all pcn-k8s containers are in running state
kubectl get pods --all-namespaces

#5. Reenable DNS
kubectl -n kube-system scale --replicas=1 deployment/kube-dns
```

10.5.2 Inspect cube status inside pcn-k8s

`pcn-k8s` is deployed as container in each node, sometimes it is helpful to inspect the cube(s) status within the container for debugging or other purposes. You can login into each node where the pcn-k8s container is running and get the information via *polycubectl* command locally.

A more convenient way to do that is using kubectl in k8s master node, first identify the name of pcn-k8s pod running in a particular node you are intereted by executing the following command:

:: kubectl get pods -n kube-system -o wide

You should see somthing like:

```
NAME                                READY   STATUS             RESTARTS   AGE   IP
↪        NODE       NOMINATED NODE   READINESS GATES
kube-proxy-dbjm6                    1/1     Running            0          28d   192.168.
↪122.200   dev-ws12   <none>           <none>
kube-proxy-stlsc                    1/1     Running            0          28d   192.168.
↪122.201   dev-ws13   <none>           <none>
kube-scheduler-dev-ws11             1/1     Running            1          28d   192.168.
↪122.199   dev-ws11   <none>           <none>
polycube-8k25h                      2/2     Running            0          25d   192.168.
↪122.200   dev-ws12   <none>           <none>
polycube-etcd-559fb856db-77kmr      1/1     Running            0          28d   192.168.
↪122.199   dev-ws11   <none>           <none>
polycube-sddh5                      2/2     Running            0          25d   192.168.
↪122.201   dev-ws13   <none>           <none>
polycube-zrdpx                      2/2     Running            0          25d   192.168.
↪122.199   dev-ws11   <none>           <none>
```

The pod name with prefix `polycube-` is pcn-k8s pod, there are a few of them in the output but only one for each node. Let's assume you want to inspect the pcn-k8s in node dev-ws13, the following command can be executed in k8s master node

```
kubectl exec -it polycube-sddh5 -n kube-system -c polycube-k8s polycubectl show cubesv
```

Here is the output for example,

```
k8sfilter:
 name  uuid                                  service-name  type  loglevel  shadow  span
↪ ports
 k8sf  6258accd-c940-4431-947c-e7292d147447  k8sfilter     TC    INFO      false   false
↪ [2 items]

k8switch:
 name       uuid                                  service-name  type  loglevel  shadow
↪span   ports
 k8switch0  c058b8fb-0e57-4ff6-be4d-5f3e99e71690  k8switch      TC    TRACE     false
↪false  [7 items]
```

10.6 Kubernetes Network Policies

`pcn-k8s` leverages on the Kubernetes Network Policies to protect the pods of your cluster from both external and internal unauthorized traffic, as well as preventing them from accessing other pods without permission.

All pods will be able to communicate with anyone as long as no policy is applied to them: this is called *Non-Isolation Mode*. As soon as one is deployed and applied to them, all traffic is going to be dropped, except for what is specified inside the policy, according to the direction it is filtering: incoming (`ingress`) or outgoing (`egress`) traffic.

Only `IP` traffic is restricted and both `TCP` and `UDP` are supported as the transport protocol. Kubernetes Network Policies are written in a `yaml` format and always include the following fields:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
```

(continues on next page)

(continued from previous page)

```
  name: policy-name
  namespace: namespace-name
```

If the `namespace` field is omitted, the policy will be deployed to the `default` namespace.

10.6.1 Select the pods

In order to select the pods to be protected, the `podSelector` field, under `spec` must be filled.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: policy-name
spec:
  podSelector:
    matchLabels:
      app: my-app
```

If a pod has a label named `app` and its value is equal to `my-app`, the above will be applied to it, regardless of its other labels.

`podSelector` only supports `matchLabels`, as `matchExpression` is not supported yet.

To select all pods the following syntax can be used:

```
spec:
  podSelector: {}
```

10.6.2 Restrict the traffic

Both incoming and outgoing traffic can be restricted. In order to restrict incoming packets, the `ingress` field must be filled:

```
spec:
  ingress:
  - from:
    # Rule 1...
  - from:
    # Rule 2...
  - from:
    # Rule 3...
```

Outgoing traffic follows a very similar pattern:

```
spec:
  egress:
  - to:
    # Rule 1...
  - to:
    # Rule 2...
  - to:
    # Rule 3...
```

NOTE: in case of restricting outgoing traffic, `policyTypes` - under `spec` - must be filled like so:

```
policyTypes:
- Ingress # Only if also restricting incoming traffic
- Egress
```

`policyTypes` can be ignored if the policy is `ingress`-only.

10.6.3 Allow external hosts

The field `ipBlock` must be filled with the IPs of the hosts to allow connections from/to, written in a `CIDR` notation. Exceptions can be optionally specified.

```
ingress:
- from:
  - ipBlock:
      cidr: 172.17.0.0/16
      except:
      - 172.17.1.0/24
```

The ips of the pods inside of the cluster are not fixed: as a result, `ipBlock` must not be used to restrict access from other pods in the cluster, but only for external entities.

10.6.4 Allow Pods

Access from other pods is restricted by using the `podSelector` - introduced earlier - and `namespaceSelector` fields. The latter works in the same fashion as the former, selecting namespaces by their label.

- In case of only using `podSelector` only the pods following the criteria specified by it and that are in the same namespace as the one specified under `metadata` will be able to access the pod.
- If only `namespaceSelector` is used, all pods contained inside the namespace with the labels specified in it will be granted access to the pod.
- For a more fine-grained selection, both can be used to specifically select pods with certain labels and that are on namespaces with specific labels.

Look at the examples in the example section to learn more about their usage.

10.6.5 Ports and protocols

In order to define protocols, one must use the `ports` field, under the `from`/`to` depending on the direction filtering:

```
ingress:
- from:
  # rule...
  ports:
  - port: 5000
    protocol: TCP
```

Refer to the examples section for more details.

10.6.6 Deploy and Remove

In order to deploy the policy, the typical `apply` command must be entered:

```
# Local policy
kubectl apply -f path-to-policy.yaml

# Remote policy
kubectl apply -f https://example.com/policy.yaml
```

To remove, one of the following commands can be issued:

```
# Delete by its path
kubectl delete -f path-to-policy.yaml

# Delete by its name
kubectl delete networkpolicy policy-name
```

10.6.7 Examples

Deny all traffic

The following policy will be applied to pods that are on namespace `production` and have label `app: bookstore`. It drops all incoming traffic.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: bookstore
ingress: []
```

The `[]` selects no pods: it drops all traffic from the specified direction.

Accept all traffic

The following policy will be applied to pods that are on namespace `production` and have label `app: bookstore`. It accepts all incoming traffic.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: bookstore
```

(continues on next page)

(continued from previous page)

```
 ingress:
 - {}
```

The {} selects everything regardless of labels: it accepts all traffic in the specified direction.

Limit connections to pods on the same namespace

The following policy will be applied to pods that are on namespace `production` and have label `app: bookstore`. It accepts connections from all pods that have labels `app: bookstore` and `role: api` and that are on the **same namespace** as the policy's one.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: bookstore
ingress:
- from:
    - podSelector:
        matchLabels:
          role: api
          app: bookstore
```

Allow connections from all pods on a namespace

The following policy will be applied to pods that are on namespace `production` and have label `app: bookstore`. It accepts connections from **all** pods that are running on namespaces that include the label `app: bookstore`.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: bookstore
ingress:
- from:
    - namespaceSelector:
        matchLabels:
          app: bookstore
```

Allow connections only from specific pods on specific namespaces

The following policy will be applied to pods that are on namespace `production` and have label `app: bookstore`. It accepts connections from pods that have label `role: api`, running on namespaces that include the label `app: bookstore`.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: bookstore
ingress:
- from:
    - podSelector:
        matchLabels:
          role: api
      namespaceSelector:
        matchLabels:
          app: bookstore
```

Rules and protocols combination

Rules are OR-ed with each other and are AND-ed with the protocols:

```
ingress:
- from:
  # Rule 1
  - ipBlock:
      cidr: 172.17.0.0/16
      except:
      - 172.17.1.0/24
  # Rule 2
  - podSelector:
      matchLabels:
        role: frontend
  ports:
  # Protocol 1
  - protocol: TCP
    port: 80
  # Protocol 2
  - protocol: TCP
    port: 8080
```

In the example above, a packet will be forwarded only if it matches one of the following conditions:

- `Rule 1` **AND** `Protocol 1`
- `Rule 1` **AND** `Protocol 2`
- `Rule 2` **AND** `Protocol 1`

- `Rule 2` **AND** `Protocol 2`

If none of the above applies, the packet is dropped.

Outgoing traffic

All the above policies can also apply to outgoing traffic by replacing `ingress` with `egress` and `from` with `to`. Also, `policyTypes` must be filled accordingly.

NOTE: keep in mind that as soon as the policy is deployed, all unauthorized traffic will be dropped, and this includes DNS queries as well! So, a rule allowing DNS queries on port 52 can be specified to prevent this.

10.6.8 Resources

For additional information about the Kubernetes Network Policies, please refer to the official documentation.

10.7 Polycube Policies

pcn-k8s CNI implementes both *standard kubernetes networking policy* and advanced Polycube networking policies. The latter provide a more flexible, simpler and more advanced approach to filter the traffic that a pod should allow or block. They include all the features from Kubernetes Network Policies and present some additional features, which are going to be described here.

The isolation mode, a core concept in Kubernetes Network Policies, is followed by Polycube Network Policies, as well: pods allow all communication until a policy - either one - is applied to them, and at that moment they will start to allow only what's explicitly specified in the policy.

10.7.1 Human Readable Policies

One of the goals of the Polycube Policies is to also encourage operators to write more effective policies, and the very first thing that can help this process is a friendly and understandable syntax.

In Polycube Policies, there is no need to use the `[]` and `{}` operators in multiple parts of the policy to select no/all pods: two convenient fields - `dropAll` and `allowAll` - are there just for this purpose, leaving no room for confusion.

```
apiVersion: polycube.network/v1beta
kind: PolycubeNetworkPolicy
metadata:
  name: deny-all
applyTo:
  target: pod
  withLabels:
    role: db
spec:
  ingressRules:
    dropAll: true
```

As shown, Polycube policies follow the natural language that humans speak - humans with a basic understanding of the English language, that is. As a matter of fact, when selecting the pods that the policy must be applied to, the `applyTo` field must be used, and the `target` field clearly specifies the need to protect a `pod` that has labels `role: db`, as finally stated in the `withLabels` field.

If you want to apply a policy to all pods on a given namespace, the following syntax may be used:

```
applyTo:
  target: pod
  any: true
```

10.7.2 Automatic type detection

There is no need to specify the policy type - `Ingress` or `Egress` - as it is automatically recognized, based on their existence in the policy YAML file.

10.7.3 Prevent Direct Access

if you want your pods to be contacted only through their Service(s) and block direct access, set the `preventDirectAccess` to `true`.

```
apiVersion: polycube.network/v1beta
kind: PolycubeNetworkPolicy
metadata:
  name: pod-drop-all
priority: 3
ingressRules:
  preventDirectAccess: true
  rules: ...
```

From now on, the pods will only allow connections that were made through their `ClusterIP`.

10.7.4 Explicit Priority

Priority can explicitly be declared by properly writing it in the policy, where a lower number indicates its importance in rules insertion.

It can be done by setting the `priority` field:

```
apiVersion: polycube.network/v1beta
kind: PolycubeNetworkPolicy
metadata:
  name: pod-drop-all
priority: 3
```

Since more recent policies always take priority (and thus are checked first), setting and explicit priority can help in those situations where you want to a policy to be checked before others.

Just to make an example: if you'd like to temporarily block all traffic to check for anomalies, there is no need for you to remove all existing policies and deploy one that drops all traffic, as you can simply give the latter a higher priority (i.e. 1) and deploy it: the higher priority will make it the first one to be checked and, as a result, all traffic would be blocked without modifying the other policies.

10.7.5 Strong distinction between the internal and external

The rules that can be specified are divided by what is internal to the cluster and what is outside.

This is done to prevent the clear bad behavior of using `IPBlock` to target pods. Peers are divided in two groups: `pod` and `world`.

The internal world can be specified like so:

```
ingressRules:
  rules:
    - action: allow
      from:
        peer: pod
        withLabels:
          role: api
        onNamespace:
          withNames:
            - beta
            - production
```

Once again, the syntax closely resembles a natural spoken language.

In Kubernetes Network Policies, namespaces can be targeted only by the labels they have: when wanting to target them, the operator is forced to assign labels to namespaces even if they just need to target very few of them. As the policy above shows, Polycube Policies provide a way to select namespaces by their names as well, while also providing the ability to do so by their labels.

The external world, instead, can be restricted by writing `world` in the `peer` field.

```
ingressRules:
  rules:
    - action: drop
      from:
        peer: world
        withIP:
          - 100.100.100.0/28
    - action: allow
      from:
        peer: world
        withIP:
          - 100.100.100.0/24
```

So, there is no need to write `exceptions`, as in Kubernetes Network Policies, because Polycube policies also have a clear distinction between actions.

10.7.6 Drop or Allow actions

```
ingressRules:
  rules:
  - from:
      peer: pod
        withLabels:
          role: api
       action: drop
```

To allow a connection, the actions that can be written are `allow`, `pass`, `forward` and `permit`.

The same applies when blocking connections, and the following words can be used: `block`, `drop`, `prohibit` and `forbid`.

The presence of multiple words to define a single action has been done to aid the definition of a policy, allowing for a more flexible semantic that is easier to remember.

This will help you create `Blacklist`-style policies by creating two or more policies: one, with a lower priority, that allows all pods in a certain port/protocol, and another one (or more) that will work as a blacklist of pods banned (i.e. those that are in the `beta` namespace).

This was a clear example of the flexibility of the Polycube Policies, but one must take very careful steps when creating a blacklist policy: although this could introduce some benefits, like lighter firewalls, it could also add some subtle inconsistencies and errors if are not created mindfully, like wrongly allowing pods to start connections.

10.7.7 Service-aware policies

Consider the following Polycube policy:

```
apiVersion: polycube.network/v1beta
kind: PolycubeNetworkPolicy
metadata:
  name: service-allow-api
applyTo:
  target: service
  withName: database
spec:
  ingressRules:
    rules:
      - from:
          peer: pod
          withLabels:
            role: api
        action: allow
```

By writing `service` as a `target`, Polycube will be aware of the fact that pods have a service applied to them and will make all the necessary steps to protect the pods according to it.

Supposing that service named `database` has `80` and `443` as `targetPorts` with protocol `TCP`, all the pods that apply such service will accept connections from pods that have label `role: api`, but only on the aforementioned ports and protocol.

This serves both as a convenient method for targeting pods without specifying the labels - `withName: database` can be seen as a clear shortcut in this case - and without specifying the ports as well.

Being `service-aware` means that firewalls will react to `Service` events, too: if later, the cluster's needs change and only the more secure `443` port is decided to be supported, the service can be updated to reflect this change and the solution will react as well by removing the behavior it used to apply for port `8080`.

The service-aware functionality is made for those particular use cases when a pod does not need a more advanced rule filtering, like allowing a pod on a certain port and allowing others on another one: as already mentioned, this is a convenient method for specifying all ports at once, and if such scenario is needed, it must be done by specifying `pod` as the peer instead of using `service`.

As a final note, only services with selectors are supported: services without selectors need to be selected by writing `world` as the peer.

10.8 Information for developers

10.8.1 Controllers

Controllers are entities that are in charge of providing you with the resource that you need, as well as watch for events and notify when one has occurred. In Polycube, five controllers are implemented:

- `Kubernetes Network Policies`
- `Polycube Network Policies`
- `Services`
- `Namespaces`
- `Pods`

Not all of them provide the same functionalities and filtering criteria, but all work based on the same principle.

Usage

The usage is inspired by Kubernetes' API style. To use the controllers, you simply need to import the `pcn_controllers` package and call the controller you'd like to use. Take a look at the following examples.

```
package main
import (
  // importing controllers
  pcn_controllers "github.com/polycube-network/polycube/src/components/k8s/pcn_k8s/
↪controllers"
)

func main() {
      // Namespaces
      namespaces, err := pcn_controllers.Namespaces().List(...)

      // Pods
      unsubscriptor, err := pcn_controllers.Namespaces().Subscribe(...)

     // etc...
}
```

Queries

All controllers can retrieve resources from the Kubernetes cache, based on some criteria. To define criteria, you must define the query: the definition is in package `pcn_types`:

```
// ObjectQuery is a struct that specifies how the object should be found
type ObjectQuery struct {
      By     string
      Name   string
      Labels map[string]string
}
```

Take a look at the following examples:

```
// I want resources that are named "my-service"
serviceQuery := pcn_types.ObjectQuery {
  By: "name",
  Name: "my-service",
}

// I want all pods that have labels "app: my-app", and "role: database"
podQuery := pcn_types.ObjectQuery {
  By: "labels",
  Labels: map[string]string {
    "app": "my-app",
    "role": "database",
}
```

Although you can create these by hand, a convenient function exists to do this and it is specifically made for use with the controllers:

```
import (
  "github.com/polycube-network/polycube/src/components/k8s/utils"
)

// ...

// Build a "by: name" query
serviceQuery := utils.BuildQuery("my-service", nil)

// Build a "by: labels" query
podQuery := utils.BuildQuery("my-service", map[string]string {
      "app": "my-app",
      "role": "database",
})

// Build a query to get all resources, regardless of name and labels
allResources := utils.BuildQuery("", nil)
```

This function returns a **pointer** to the actual query structure because that's what controllers need. When wanting to get all resources, the function returns nil, so you may even just use a nil value without calling the BuildQuery function.

List resources

To list resources, you need to first create the queries, and then call the **List** function of the controller. Not all controllers support both name and label criteria: i.e. the Pod Controller only supports labels.

```
// I want all services that apply to pods with labels "app: my-app" and "role: db"
// and are on a namespace called "production"
// So, first create the queries for both the service and namespace.
serviceQuery := utils.BuildQuery(nil, map[string]string {
      "app": "my-app",
      "role": "db",
})

nsQuery := utils.BuildQuery("production", nil)
// Then, get them. Note: there might be more than one service which applies to those␣
↪pods.
servicesList, err := pcn_controllers.Services().List(serviceQuery, nsQuery)
if err != nil {
  return
}
for _, service := range servicesList {
  // Do something with this service...
}
```

So, usually, the first argument is criteria about the resource, while the second is reserved for criteria about the namespace where you want to find such resources.

To give additional information:

- The `Kubernetes Network Policies` and `Polycube Network Policies` controllers only support querying the policy by name
- The `Pod` controller only supports querying by labels
- The `Pod` controller also supports a third argument for the node where you want this pod to be located.
- The `Services` controller supports both name and labels, but when using labels it searches for them in the **spec.selector** field, not those under its metadata.
- The `Namespaces` controller work with namespaces, which cannot belong to other resources and only want one argument.

Note that, according to the criteria, it may take you a long time to get the results. Whenever possible, or when you expect a query to return lots of resources, adopt an async pattern or use multiple goroutines.

Watch for events

To watch for events, you need to use a controller's `Subscribe` function by passing to it the event type you want to monitor, the resource criteria, and the function to be executed when that event is detected.

```
func firstfunc() {
  // I want to "myfunc" to be notified whenever a new pod is born.
  // Pod controller has the most complex subscribe function, as it also asks you for the␣
↪phase of the pod.
  unsub, err := pcn_controllers.Pods().Subscribe(pcn_types.New, nil, nil, &pcn_types.
↪ObjectQuery{Name: "node-name"}, pcn_types.PodRunning, myfunc)
```

(continues on next page)

(continued from previous page)

```
  // ...

  // I am not interested in that event anymore
  unsub()
}

func myfunc(currentState, previousState *core_v1.Pod) {
  // Do something with it...
}
```

As the above example shows, the `Subscribe` function returns a pointer to a function that you need to call when you're not interested in that event anymore.

The function to execute must always have two arguments: the current state of the object and its previous state. There are three event types: `New`, `Update`, `Delete`.

Just some heads up:

- When monitoring `New` events, only the current state of the object is present, the previous is obviously always `nil`.
- When monitoring `Delete` events, the object does not exist anymore, so the current state is always `nil`.

All the `Subscribe` functions share a similar structure to the `List` function in the same controller, to make sure about their usage, check their definitions in the `pcn_controllers` package

10.8.2 Creating the Docker Images

Docker 18.06 is needed to build the images, and the daemon has to be started with the *–experimental* flag. See this issue to have more information.

```
export DOCKER_BUILDKIT=1 # flag needed to enable the --mount option
docker build --build-arg DEFAULT_MODE=pcn-k8s -t name:tag .
docker push name:tag
```

CHAPTER

ELEVEN

POLYCUBE SERVICES

This folder contains the list of services (a.k.a. *cubes*) currently available in *polycube*.

11.1 Helloworld

Note: documentation of the current release is focused on the final user and not in developers. This service is intended to be used only by developers, so this documentation could be incomplete and have some inacuracies.

This service demonstrates how to create a minimal cube, which includes the dataplane *fast path*, the *slow path*, running in user-space, and the control/management portions (e.g., to configure the service).

Helloworld is a simple service that receives the traffic on a network interface and can either:

- send packets to a second interface
- send packets to the slow path
- drop packets

The behavior of this service can be changed by setting the `action` flag, which tells the data plane how the packets have to be processed.

11.1.1 How to use

```
# create network namespaces
# DO IT YOURSELF

# create the instance
polycubectl helloworld add hw0

# add ports (only two are supported)
polycubectl hw0 ports add port1 peer=veth1
polycubectl hw0 ports add port2 peer=veth2

# change action
polycubectl hw0 set action=drop

# send packets to the service

# try another actions, forward, slowpath
polycubectl hw0 set action=forward
```

11.1.2 Service structure

Helloworld includes the minimum amount of code that a service requires to be run.

- **src/helloworld_dp.h** contains the eBPF code of the cube. It is not necessary to define the code in a separated file, however we suggest it to keep things organized.
- **src/helloworld.[h,cpp]** contain the definition of the main class of the service. This class represents an instance of the service and implements the methods for creating and destroying cubes, adding and removing ports from it. Finally, it implements the control plane that allows to change the behavior of the cube by updating values in the eBPF maps.
- **src/api** contains the rest api implementation, these files are automatically generated and do not have to be modified by the programmer.
- **src/[default-src, interface, serializer]** contain different pieces of the services that do not need to be modified by the developer.
- **src/helloworld-lib.cpp** contains the implementation of interface that is used when the service is compiled as a shared library.
- **datamodel/helloworld.yang** contains the service datamodel.

11.1.3 Compile and install

By default services shipped with *polycube* are compiled and installed all together.

The instructions below are provided for people who want to compile and install this service separately, which may be useful in case we want to create our own service and we start from HelloWorld as minimal skeleton.

In order to compile and install a service run the following commands:

```
# in the service folder
mkdir build; cd build
cmake ..
make
sudo make install
```

The shared library implementation is installed in the default libraries path of the host.

11.2 Simple Bridge

This services implements a fast and simple Ethernet bridge.

11.2.1 Features

- Up to 1024 hosts
- Old entries are automaticall pruned from filtering database

11.2.2 How to use

Please see the examples below:

Examples

Example 1

In this example two network namespaces are connected together by a simple bridge instance.

The following code configures the network namespaces and virtual network interfaces to be used.

```
# copy and paste in your terminal

# namespace ns1 -> veth1 10.0.0.1/24
# namespace ns2 -> veth2 10.0.0.2/24

for i in `seq 1 2`;
do
    sudo ip netns del ns${i} > /dev/null 2>&1 # remove ns if already existed
    sudo ip link del veth${i} > /dev/null 2>&1

    sudo ip netns add ns${i}
    sudo ip link add veth${i}_ type veth peer name veth${i}
    sudo ip link set veth${i}_ netns ns${i}
    sudo ip netns exec ns${i} ip link set dev veth${i}_ up
    sudo ip link set dev veth${i} up
    sudo ip netns exec ns${i} ifconfig veth${i}_ 10.0.0.${i}/24
done
```

Create a simple bridge instance, add and connects ports to virtual interfaces

```
# create instance
polycubectl simplebridge add br0

# add and connect port to veth1
polycubectl br0 ports add toveth1 peer=veth1

# add and connect port to veth2
polycubectl br0 ports add toveth2 peer=veth2
```

Ping between namespaces

```
# ping ns1 from ns2
sudo ip netns exec ns2 ping 10.0.0.1
```

Print whole `br0` status

```
polycubectl br0 show
```

Delete ports

```
polycubectl br0 ports del toveth1
polycubectl br0 ports del toveth2
```

Remove `br0`

```
polyubectl del br0
```

11.3 Bridge

`pcn-bridge` is an 802.1Q Ethernet bridge.

11.3.1 Features

- Support for VLANs
- Support for access and trunk mode for the ports.
- Support for CSTP (Common-STP) and PVSTP (Per Vlan-STP)

11.3.2 Limitations

- Currently it does not accept all vlans on a trunk port

11.3.3 How to use

Create instances and ports

```
# create the instance
polycubectl bridge add br1

# add ports
polycubectl br1 ports add p1
```

VLAN configuration

```
# change VLAN in access mode
polycubectl br1 ports p1 access set vlanid=2

# change port mode (access/trunk)
polycubectl br1 ports p1 set mode=trunk

# add an allowed vlan in a trunk port
polycubectl br1 ports p1 trunk allowed add 10

# change native vlan in a trunk port
polycubectl br1 ports p1 trunk set native-vlan=2

# enable/disable native vlan in a trunk port
polycubectl br1 ports p1 trunk set native-vlan-enabled=false
```

Spanning Tree configuration

```
# enable/disable spanning tree protocol
polycubectl br1 set stp-enabled=true

# view active instances of STP
polycubectl br1 stp show

# view a particular instance
polycubectl br1 stp 1 show

# modify a parameter in an active instance
polycubectl br1 stp 1 set priority=28672

# view STP configuration in a port
polycubectl br1 ports p1 stp 1 show

# modify a paremeter in an active instance of a port
polycubectl br1 ports p1 stp 1 set port-priority=64
```

Examples

Example 1 - Connectivity

In this example two network namespaces will be connected together by a bridge instance.

The following code configures the network namespaces and virtual network interfaces to be used.

```
# copy and paste in your terminal

# namespace ns1 -> veth1 10.0.0.1/24
# namespace ns2 -> veth2 10.0.0.2/24

for i in `seq 1 2`;
do
    sudo ip netns del ns${i} > /dev/null 2>&1 # remove ns if already existed
    sudo ip link del veth${i} > /dev/null 2>&1

    sudo ip netns add ns${i}
    sudo ip link add veth${i}_ type veth peer name veth${i}
    sudo ip link set veth${i}_ netns ns${i}
    sudo ip netns exec ns${i} ip link set dev veth${i}_ up
    sudo ip link set dev veth${i} up
    sudo ip netns exec ns${i} ifconfig veth${i}_ 10.0.0.${i}/24
done
```

Create a bridge instance, add and connects ports to virtual interfaces

```
# create instance
polycubectl bridge add br1

# add and connect port to veth1
polycubectl br1 ports add toveth1 peer=veth1
```

(continues on next page)

(continued from previous page)

```
# add and connect port to veth2
polycubectl br1 ports add toveth2 peer=veth2
```

Ping between namespaces

```
# ping ns2 from ns1
sudo ip netns exec ns1 ping 10.0.0.2
```

Print whole `br1` status

```
polycubectl br1 show
```

Delete ports

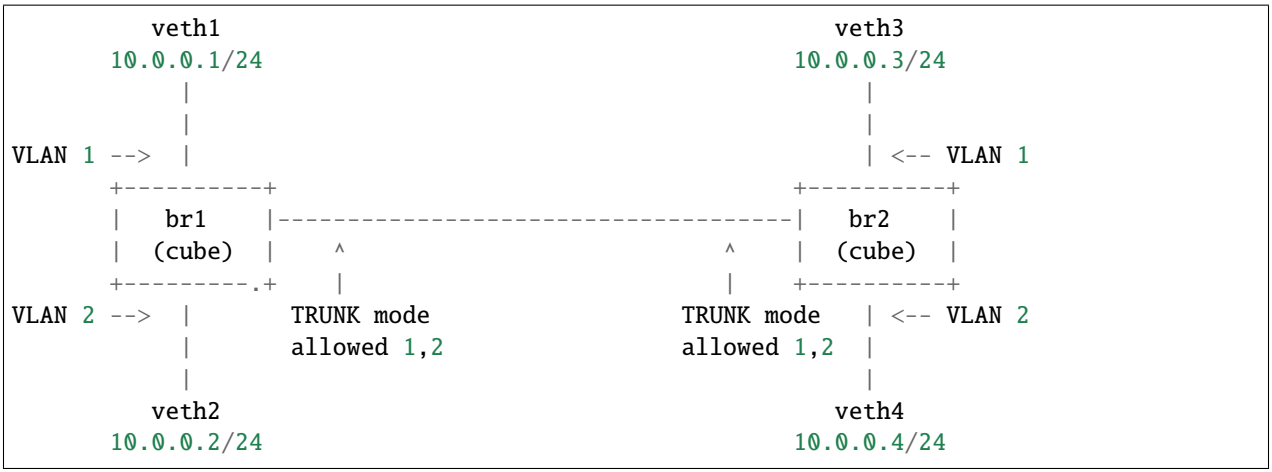
```
polycubectl br1 ports del toveth1
polycubectl br1 ports del toveth2
```

Remove `br1`

```
polyubectl del br1
```

Example 2 - VLAN

In this example we will test the VLAN support. We will configure two bridges, and four network namespaces connected to them.



The following code configures the network namespaces and virtual network interfaces to be used.

```
# copy and paste in your terminal

# namespace ns1 -> veth1 10.0.0.1/24
# namespace ns2 -> veth2 10.0.0.2/24
# namespace ns3 -> veth3 10.0.0.3/24
# namespace ns4 -> veth4 10.0.0.4/24

for i in `seq 1 4`;
do
```

(continues on next page)

(continued from previous page)

```
    sudo ip netns del ns${i} > /dev/null 2>&1 # remove ns if already existed
    sudo ip link del veth${i} > /dev/null 2>&1

    sudo ip netns add ns${i}
    sudo ip link add veth${i}_ type veth peer name veth${i}
    sudo ip link set veth${i}_ netns ns${i}
    sudo ip netns exec ns${i} ip link set dev veth${i}_ up
    sudo ip link set dev veth${i} up
    sudo ip netns exec ns${i} ifconfig veth${i}_ 10.0.0.${i}/24
done
```

Create bridge instances, and connect virtual interfaces to them

```
# create instances
polycubectl bridge add br1
polycubectl bridge add br2

# create ports on br1
polycubectl br1 ports add toveth1 peer=veth1
polycubectl br1 ports add toveth2 peer=veth2
polycubectl br1 ports add tobr2 mode=trunk

# create ports on br2
polycubectl br2 ports add toveth3 peer=veth3
polycubectl br2 ports add toveth4 peer=veth4
polycubectl br2 ports add tobr1 mode=trunk

# connect the two bridges
polycubectl connect br1:tobr2 br2:tobr1
```

Configure VLANs

```
# By default, ports are configured in access mode, with VLAN 1
# Instead, ports in trunk mode have VLAN 1 allowed by default
# (and that is also the native vlan)

# br1
polycubectl br1 ports toveth2 access set vlanid=2
polycubectl br1 ports tobr2 trunk allowed add 2

# br2
polycubectl br2 ports toveth4 access set vlanid=2
polycubectl br2 ports tobr1 trunk allowed add 2
```

Ping between namespaces

```
# ping ns3 from ns1
sudo ip netns exec ns1 ping 10.0.0.3 # ok

# ping ns4 from ns2
sudo ip netns exec ns2 ping 10.0.0.4 # ok
```

(continues on next page)

(continued from previous page)

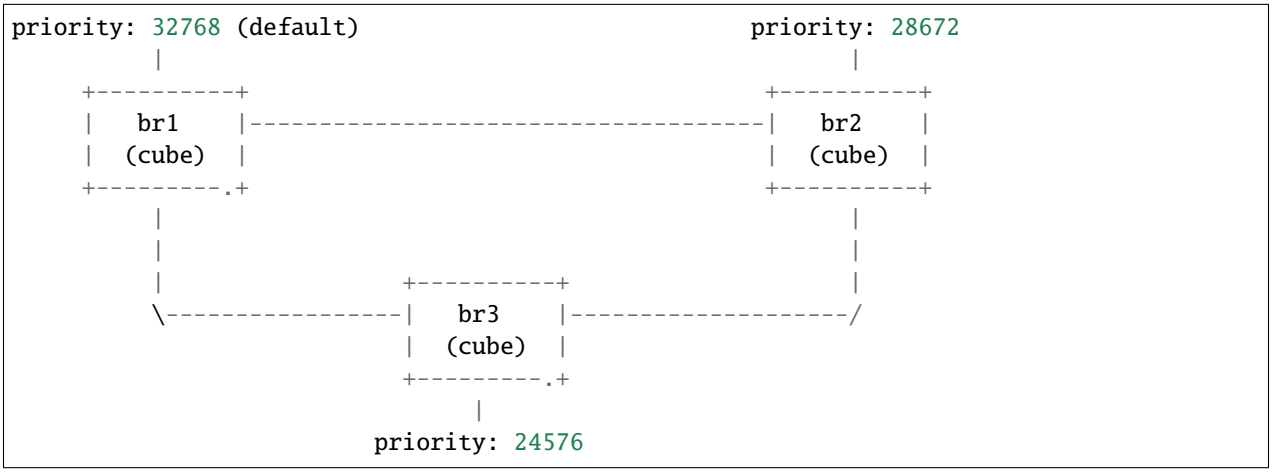
```
# ping ns4 from ns1
sudo ip netns exec ns1 ping 10.0.0.4 # packet discarded by br2: not the same VLAN!
```

Delete bridges

```
polycubectl br1 del
polycubectl br2 del
```

Example 3 - Spanning Tree

In this example we will test the Spanning Tree configuration. We will have three bridges connected each other in a triangle.



In this configuration, we expect that bridge `br3` will be the root bridge (lower priority). Furthermore, according to the STP algorithm, the port that should be blocked is port `br1:tobr2`.

Create bridge instances and connect them each other

```
# create instances
polycubectl bridge add br1
polycubectl bridge add br2
polycubectl bridge add br3

# add ports
polycubectl br1 ports add tobr2
polycubectl br1 ports add tobr3
polycubectl br2 ports add tobr1
polycubectl br2 ports add tobr3
polycubectl br3 ports add tobr1
polycubectl br3 ports add tobr2

# connect ports
polycubectl connect br1:tobr2 br2:tobr1
polycubectl connect br1:tobr3 br3:tobr1
polycubectl connect br2:tobr3 br3:tobr2
```

Enable STP in each bridge

```
polycubectl br1 set stp-enabled=true
polycubectl br2 set stp-enabled=true
polycubectl br3 set stp-enabled=true
```

Change priority of bridges

```
# In each bridge, STP instance of VLAN 1 is active by default
# (all the ports are configured by default in access mode with VLAN 1)

# Default bridge priority: 32768
polycubectl br2 stp 1 set priority=28672
polycubectl br3 stp 1 set priority=24576

# Wait for convergence
sleep 50
```

Check ports

```
# br1
polycubectl br1 ports tobr2 stp 1 show state # blocking
polycubectl br1 ports tobr3 stp 1 show state # forwarding

# br2
polycubectl br2 ports tobr1 stp 1 show state # forwarding
polycubectl br2 ports tobr3 stp 1 show state # forwarding

# br3
polycubectl br3 ports tobr1 stp 1 show state # forwarding
polycubectl br3 ports tobr2 stp 1 show state # forwarding
```

Delete bridges

```
polycubectl br1 del
polycubectl br2 del
polycubectl br3 del
```

11.4 Simple Forwarder

Simple forwarder is another sample service, in this case it is more advanced than the HelloWold service, it is used both, for internal testing and to show the usage of more complex features.

11.4.1 Description

Simple forwarder contains a primitive forwarding table that according to the packet's ingress port it can either: - send the packet to another port - send the packet to the slow path - drop the packet

11.4.2 How to use

TODO: api is document in that way, how to do it?

The api for the forwarder service can be found [here](https://app.swaggerhub.com/apis/netgrp-polito/forwarder-api/1.0.0). The forwarding table of the service can be modified through the *.../actions/* api, it allows to add an action for each port.

11.5 Policy-Based Forwarder

This service drops, forwards, or sends to the slowpath each packet that matches one of the defined rules, based on the source and destination MAC addresses, VLAN ID, IPv4 addresses, level 4 protocol and ports, and on the input port. Policy rules can include one or more of the above fields; if a given field is missing, its content is non-influent for the matching.

11.5.1 Features

Supported features:

- Wildcard matching on the above fields:
  - Input port
  - MAC source/destination
  - VLAN ID
  - IPv4 source/destination
  - L4 protocol (TCP/UDP)
  - L4 source/destination port
- Possible actions:
  - Forward packet on a given output port
  - Send packet to slow path
  - Drop packet

11.5.2 Limitations

- Rules numbering - Rules number have to be in sequence, starting from 0. For example, a valid rule set has rules with ID 0, 1, 2, ... while a set with rules 0, 1, 3 will not match the rule number 3.
- Unsupported features: - More complex actions (e.g., modify field, push/pop VLAN tags) - Any field beyond the above list
- The code does not recognize the situation in which a complex rule is deleted in such a way that the remaining rules can leverage a more aggressive optimization (e.g, checking only on L2/L3 fields). Hence, in this case the optimization is not applied.

11.5.3 How to use

The [example](./example) show how to use this service to create a multi-hop forwarding service among different namespaces. This could be used to see the basic commands of the `pcn-pbforwarder` service.

11.5.4 Implementation details

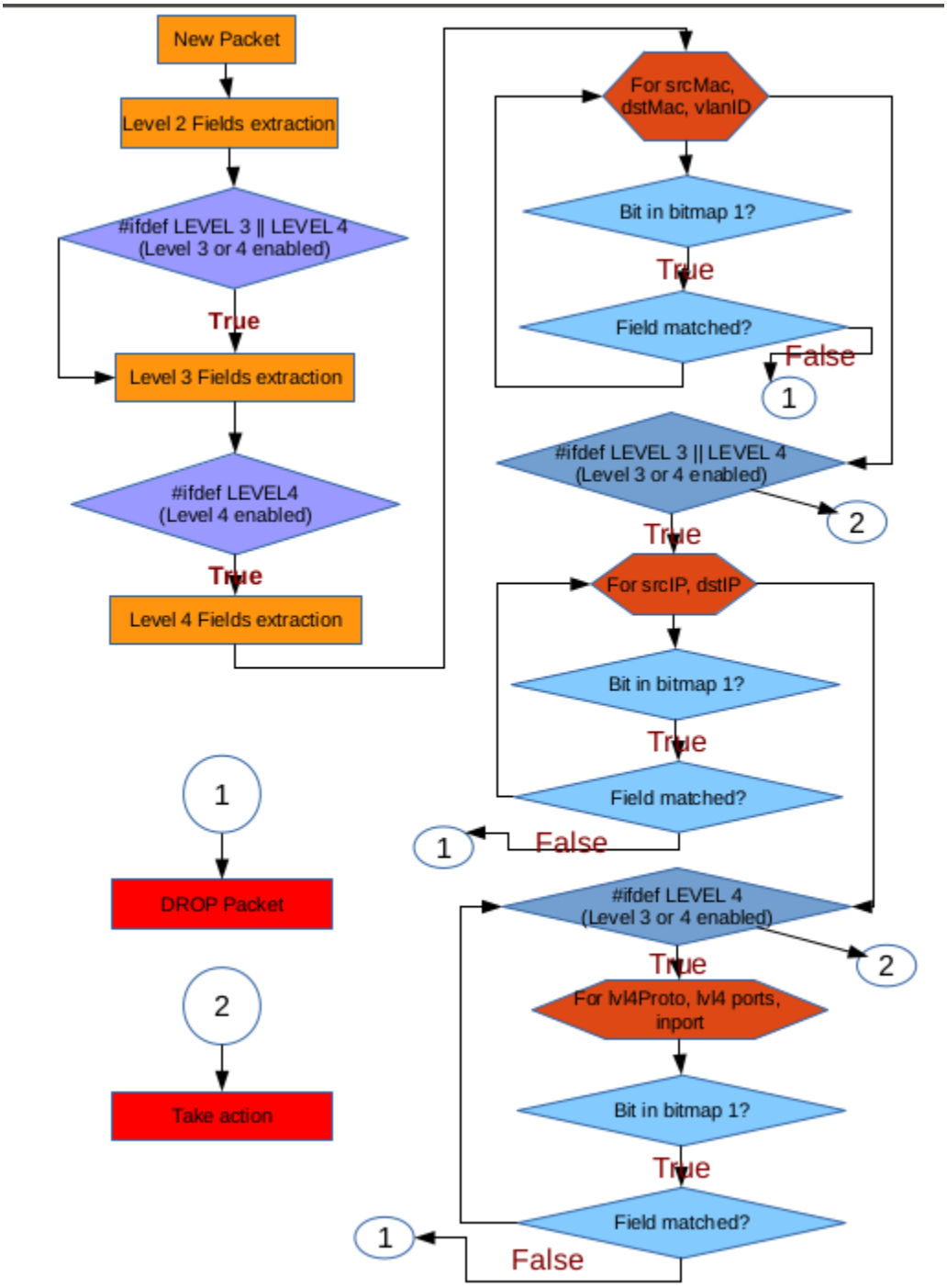
Data plane - fast path

The code of the data path is complicated by the fact that currently eBPF does not support maps with ternary values (i.e., wildcard maps). Therefore the algorithm is forced to implement a linear search, checking if the incoming packet matches the first rule, if not it moves to the second rule, and so on.

The matching is done by using a flag, associated to each rule, whose bits are `1` in correspondence of the fields that are valid in the rule, and `0` if the rule does not consider that field and hence the content of the field can be simply ignored. This introduces a limitation in the current number of rules that can be matched, as the eBPF has to unroll a loop in order to check all the rules, hence quickly reaching the maximum number of instructions.

In order to (partially) overcome the above limitation, the current data plane code depends on two macros: `LEVEL` and `RULES`, both defined by `generate_code()` in `Pbforwarder.cpp`. Based on the `LEVEL` macro, the fast path enables only the portions of the code that are strictly needed for the matching, based on the fields that actually need to be checked. In order words, if the rules refer only to L2 fields, the fast path enables only the portion of code that handles L2 processing, avoiding unnecessary checks on L3 and L4 fields. In fact, the `RULES` macro keeps the actual number of configured rules, hence it represent the number of times the loop is unrolled by the eBPF compiler.

The flowchart below summarizes the fast path algorithm of the service, which is split in multiple dataplane files (source folder).

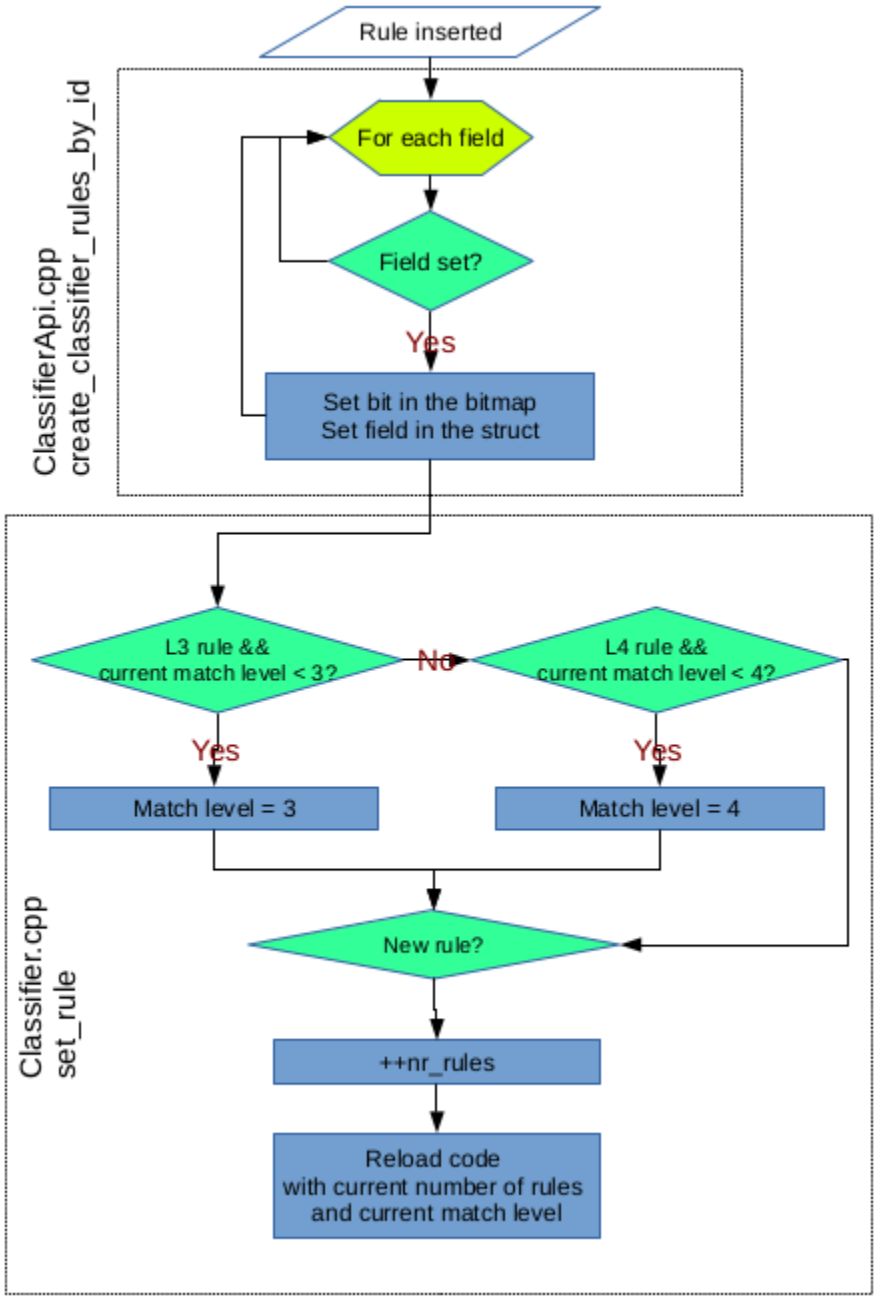


This service does not have any noticeable slow path algorithm for the data plane, which is then omitted here.

Control path

When a rule is inserted, the corresponding API method is called in `DpforwarderApi.cpp`. This prepares a struct and the bitmap based on the input received from the user and passes the control to the set_rule method in `Dpforwarder.cpp`. The set_rule method evaluates the level required by the rule and sets the attribute `match_level`, evaluates if the rule is new and sets the attribute *nr_rules*, then invokes *generate_code()* to prepare the eBPF code and loads it.

When the number of rules exceeds the capacity of the data plane, the eBPF compiler returns an error and the user is warned that his latest rule cannot be applied.



11.6 Firewall

This service implements a transparent firewall. It can be attached to a port or a netdev, and it may drop or forward each packet that matches one of the defined rules, based on the source and destination IPv4 addresses, level 4 protocol and ports, and TCP flags. Policy rules can include one or more of the above fields; if a given field is missing, its content does not influence the matching.

Non-IP packets are always accepted and forwarded, without any check.

11.6.1 Features

Supported features:

- Matching on following fields:
  - IPv4 source/destination (with prefix match)
  - L4 protocol (TCP/UDP/ICMP)
  - L4 source/destination port
  - TCP Flags
  - Connection tracking status
- Possible actions:
  - Accept packet from the interface from which it was received and forward it to the other
  - Drop packet
- Non-IP packets are always accepted
- Up to 5k rules for each chain (INGRESS/EGRESS)

11.6.2 How to use

Ingress ad egress chains

The service supports independent *ingress* and *egress* policy chains, with two different policy sets:

- **ingress**: packets that come from the external world and that are try to reach the *inside* of your network (e.g., in case the firewall is attached to a network device, this refers to packets that are trying to reach your TCP/IP stack);
- **egress**: packets that come from your inside network and that are trying to reach the external world.

Rule insertion

Rule insertion is guaranteed to be *atomic*: during the computation of the new datapath, the old rule set is used until the new rule set is ready, and only at that moment the new policies will be applied.

Rules can be:

- **inserted**: the `insert` action adds your rule at the *beginning* of the ruleset (i.e., it becomes the new rule `0`; existing rules are pushed down).
- **appended**: the `append` action adds your rule at the *end* of the ruleset (i.e., it becomes the last rule of the ruleset, before the *default* rule).
- **update**: the `update` action updates a specific rule.

- **deleted**: the `delete` action deletes a specific rule.

Rule insertion is an expensive operation. For this reason, if you are using the REST interface, you can exploit different endpoints to optimize this expensive operation:

- `/insert`, `/delete`, `/append` and `PUT` on `rule/<id>` (update): these endpoints are used to perform a single operation on a rule. As soon as the rule-set is updated, it is compiled and all the modifications are immediately inserted in the datapath.
- `/batch`: as suggested by the name, this endpoint is used to perform multiple operation on a single HTTP request. Instead of compiling the new rule-set as soon as a single operation is fulfilled, it waits for all the actions described in the request to be executed. Finally, a single compilation is performed and the datapath is updated once.

Concerning the batch endpoint, it accepts a JSON list of rules like:

```
{
  "rules": [
    {"operation": "insert", "id": 0, "l4proto":"TCP", "src":"192.168.1.1/32", "dst":"192.
↪168.1.10/24", "action":"drop"},
    {"operation": "append", "l4proto": "ICMP", "src":"192.168.1.100/32", "dst":"192.168.
↪1.100/24", "action":"drop"},
    {"operation": "update", "id": 0, "l4proto":"TCP", "src":"192.168.1.2/32", "dst":"192.
↪168.1.20/24", "action":"accept"},
    {"operation": "delete", "id": 0},
    {"operation": "delete", "l4proto":"ICMP", "src":"192.168.1.100/32", "dst":"192.168.1.
↪100/24", "action":"drop"}
  ]
}
```

Each element of the `rules` array MUST contain an operation (*insert*, *append*, *update*, *delete*) plus a rule/id that represents the actual target of the above operation. All the listed operation are performed sequentially, hence the user must sent the operations with the appropriate order. Pay attention when sending some DELETE with other INSERT; you have to take in mind that during such operations IDs may vary (increase or decrease).

This features is also available from the `polycubectl` command line. It is strongly suggested to create a JSON file containing the batch of rules and then type:

`polycubectl firewall <fwname> chain <chainname> batch rules= < filename.json`

Using the redirection diamond you are able to insert the file content in the body of the HTTP POST request generated from the command.

Default action

The default action if no rule is matched is **ACCEPT**. This can be changed for each chain independently by issuing the command `polycubectl firewall fwname chain INGRESS set default=DROP` or `polycubectl firewall fwname chain EGRESS set default=DROP`.

Statistics and firewall status

The service tracks the number of packets and the total bytes that have matched each rule. Statistics can be seen by issuing the command `polycubectl firewall fw chain INGRESS stats show` (where `fw` is the name of your firewall instance); follow the help for further details. To flush all the statistics (i.e. both packets and bytes count for every rule) about a chain, issue the following command `polycubectl firewall fw chain INGRESS reset-counters`.

Additional statistics and status information can be shown with the command `polycubectl firewall fw show` (where `fw` is the name of your firewall instance); for instance, in case the connection tracking is enabled, this shows also all the TCP/UDP sessions that are currently active in the firewall.

Connection tracking and stateful operations

This firewall supports stateful operations, e.g., it allows a to set a ACCEPT rule for a given traffic in a given direction (e.g., allow incoming connection on port 22, to enable reaching your local SSH server), and automatically accept also the packets that are generated in the opposite direction and that relate to the above rule.

The connection tracking is enabled by default; its status can be inspected with command `polycubectl firewall fw show`, which shows also the status of all the TCP/UDP sessions that are currently active in the firewall. This behavior can be changed with the command `polycubectl fw1 set accept-established=OFF`.

Connection tracking can still be used, even if the global command apparently set it to OFF, by selectively enabling this feature on a given subset of traffic. For instance, the above command:

```
polycubectl fw1 chain EGRESS append l4proto=TCP sport=22 conntrack=ESTABLISHED␣
↪action=ACCEPT
```

will accept all TCP packets that come from source port 22 (i.e., a local SSH server) and whose connection status is ESTABLISHED. This means that a packet had to be received by your host on port 22, your local server has accepted the connection, hence the packets generated in the opposite direction (i.e., EGRESS) are accepted.

11.6.3 Examples

First simple examples: enabling SSH connection to your host

Here there is a simple (but complete) example, which allows a given machine:

- to connect to the Internet and browse HTTPS sites (and nothing else)
- to accept SSH connections from the Internet (and nothing else)
- to resolve DNS names (UDP port 53 is enabled in both directions)

We assume that the machine has a network card named `enp0s3`.

```
# Create firewall
polycubectl add firewall fw1

# Attach firewall to the network card (enp0s3)
```

(continues on next page)

(continued from previous page)

```
polycubectl attach fw1 enp0s3

# Set default action to DROP for both INGRESS and EGRESS chains
polycubectl fw1 chain INGRESS set default=DROP
polycubectl fw1 chain EGRESS set default=DROP

# Enable incoming connections on port 22 (to ssh to my server from the external world)
polycubectl fw1 chain INGRESS append l4proto=TCP dport=22 action=ACCEPT

# Enable outgoing connections on port 443 (to connect to HTTPS servers from my machine)
polycubectl fw1 chain EGRESS append l4proto=TCP dport=443 action=ACCEPT

# Enable port 53 in both directions (to enable name resolution)
polycubectl fw1 chain INGRESS append l4proto=UDP sport=53 action=ACCEPT
polycubectl fw1 chain EGRESS append l4proto=UDP dport=53 action=ACCEPT

# Enable established connections to go through, independently from the port they're using
# Instead of the above two commands, we can use a single default command, i.e.
#    polycubectl fw1 set accept-established=ON
polycubectl fw1 chain INGRESS append l4proto=TCP conntrack=ESTABLISHED action=ACCEPT
polycubectl fw1 chain EGRESS append l4proto=TCP conntrack=ESTABLISHED action=ACCEPT

# Show statistics for the INGRESS chain of the firewall
polycubectl fw1 chain INGRESS show

# Show general statistics for the firewall (e.g., the current ongoing sessions)
polycubectl fw1 show

# Remove the firewall
polycubectl del fw1
```

More examples

The examples source folder contains some simple scripts to show how to configure the service.

Also under the test directory, there are plenty of scripts that test the firewall using both single and batch rule insertion/deletion.

11.6.4 Implementation details

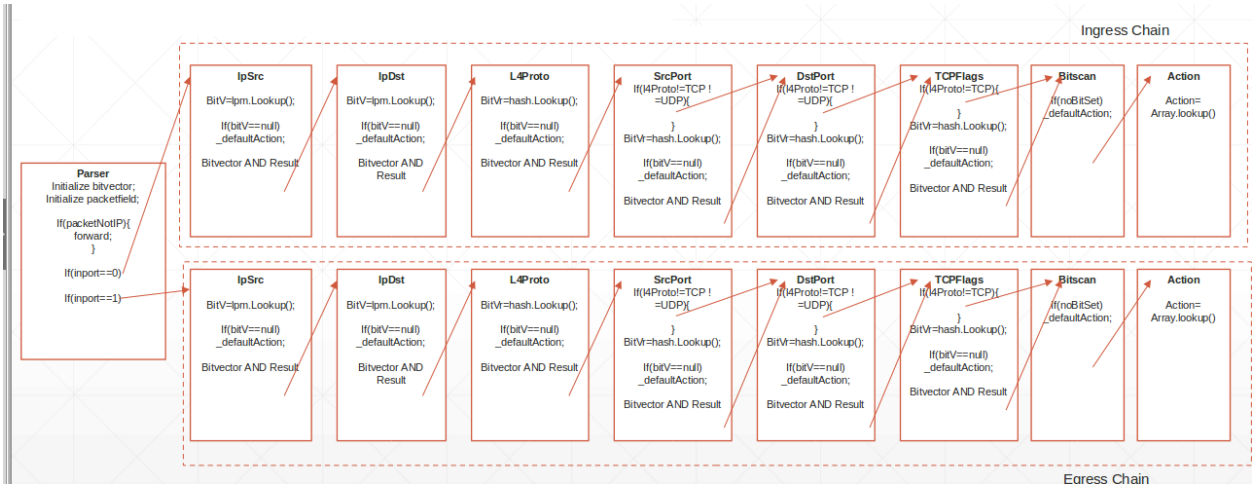
Data plane - fast path

Currently eBPF does not support maps with ternary values (i.e., *wildcard maps*), this forced to implement an algorithm that could offer this functionality and support a large number of rules, the **Linear Bit Vector Search**, that is particularly suitable to be implemented in eBPF and modularized using tail calls, but has an O(NRules) complexity.

A first module parses the packet and sends it to the ingress or egress chain. Each chain has a series of eBPF programs that evaluate one single field, compute the bit vector (in linear time) and sends the packet to the next module. The second-to-last module uses the *De Bruijn sequence* to perform a first bit set search, and based on the results calls the next module that performs the actual action on the packet.

Each module is injected only if the rule set requires it (for example, if no rule requires matching on IP source, the module in charge of doing it is not injected). The rule limit and the O(N) complexity is given by the bit vector computation, that requires a linear search of the array, performed using loop unrolling.

An overview of the algorithm is depicted in the figure below.



11.6.5 Control Plane

Code structure

The control plane is in charge of managing each eBPF module. The code has been organized hierarchically to simplify the implementation. The Firewall class acts as a master, it keeps track of all the injected modules. API calls are managed by the ChainRule and Chain classes. Each module is represented in the control plane by a class inheriting from the Program interface, and encapsulates the eBPF module management, offering uniform interfaces to inject or remove the module or interact with its tables. This structure has the advantage of masking a number of MACROS present in the bpf code that are substituted at run-time based on the configuration, for example the number of rules.

Rules computation

The Linear Bit Vector Search requires computing tables of bit vectors, where each table represent a field, each row represents a value for that field and the matched rules in the form of a bit vector (where the Nth bit is 1 if the rule is matched, 0 if not). Considering the complexity of the operation, the choice was to compute the tables from zero each time a rule is modified.

11.7 DDoS Mitigator

This service implements a DDoS Mitigator, which can drop (malicious) packets at very high speed based on a blacklist applied on either IP source or destination addresses. Instead, non-IP traffic (e.g., IPv6, ARP, etc.) is always forwarded.

11.7.1 Features

Supported features:

- blacklist of source IP addresses (`blacklist-src`)
- blacklist of destination IP addresses (`blacklist-dst`)
- `statistics` about dropped traffic

11.7.2 Limitations

- IPv6: currently unsupported; all the IPv6 traffic is forwarded as is, without any check.

11.7.3 Performance

Although this service can attach to either the `TC`, `XDP_SKB` and `XDP_DRV` eBPF hooks, we suggest to use `XDP_DRV` (if supported by your NIC driver) in order to get the highest dropping rate. This loads the service as an `XDP` program in driver mode, hence discarding packets as soon as they arrive in the NIC driver, before delivering them to the main networking components of the operating system.

To further improve the performance of this service, the code implementing the datapath is dynamically created (and re-injected in the kernel) in a way that includes only the required features. For example, the code that handles IP destination address is not present if the user does not ask for IP destination filtering. This enables this service to create a datapath program that includes only the minimum lines of code required to implement the requested features, with a positive impact on the performance of the system.

11.8 SYN Flood Monitor

This service exports some metrics that can be used to detect a possible SYN Flood attack.

11.8.1 Features

- Retrieves a set of TCP/IP parameters that can be used to detect SYN flooding attacks

11.8.2 Limitations

- It must be launched on the host that needs to be monitored. It cannot operate as a 'main in the middle' mode, i.e., inspecting network traffic directed toward a remote host.

11.8.3 How to use

Technically, `pcn-synflood` is a transparent service, hence it should be attached to an existing network interface (e.g., netdev or a virtual link between Polycube services). However, given that the current implementation retrieves traffic statistics using the metrics provided by the operating system, it can be even instantiated without attaching it to any network interface.

11.8.4 Exported metrics

- `tcpAttemptFails`: number of failed TCP connections [1].
- `tcpOutRsts`: number of TCP segments sent, containing RST flag.
- `deliverRatio`: ratio between the number of IP pkts delivered to application protocols and the total number of received pkts.
- `responseRatio`: ratio between the number of IP pkts requests to send by application protocols and the total number of received pkts.

11.8.5 Additional details

[1] `tcpAttemptFails`

It measures the number of times TCP connections have made a direct transition to the CLOSED state from either the SYN-SENT state or the SYN-RCVD state, plus the number of times TCP connections have made a direct transition to the LISTEN state from the SYN-RCVD state. It refers to variable `tcpAttemptFails` documented in RFC 1213.

It is worth mentioning that this parameter is rather general and can be used to check for unusual situations on both client and server side. For instance, it can be used to detect either (1) that a server is under attack (e.g., TCP state machine goes from SYN-RCVD to LISTEN or CLOSED), or (2) that a client is currently attacking a remote target (e.g., TCP state machine goes from SYN-SENT to CLOSED).

The most common case, i.e., that a server is under attack, corresponds at least to the following unusual TCP sequences:

- [SYN, timeout]. The server receives a SYN packet, but it cannot answer any more because it is overwhelmed. This connection will be ended after server time-out, as described earlier.
- [SYN (Client, Server), RST (Server, Client)]. This sequence means either that the server is the victim of a DoS attack because it cannot reply to the legitimate client any more, or that there is not applications listening on that port.
- [SYN, SYN/ACK, timeout]. The server waits indefinitely for the ACK packet, either because the IP source address is spoofed or because the ACK packet is rejected because of network congestion. This sequence can correspond to a DoS attack. This connection will be ended after server time-out.
- [SYN, SYN/ACK, RST]. This handshake sequence can correspond to a DDoS attack. At the reception of the SYN/ACK packet, the client host then transmits an RST packet to the server because it never sent a SYN packet.

11.9 Router

This service implements an IPv4 router.

11.9.1 Features

- IPv4 addresses and routes
- Only static routes are supported
- Up to 5 secondary addresses per interface
- Handling of ARP packets
- IPv6 and VLANs are not supported

11.9.2 How to use

Important note about MAC filtering on physical NICs

When setting up the router connected to a physical interface, remember to set the MAC address of that router interface equal to the MAC address of the physical NIC. This allows the frames that come into the pcn-router to have the same MAC address of the physical interface, which will allow the frame to go through. Otherwise, the NIC may discard incoming frames because of the wrong MAC destination address, hence the traffic will never reach the router interface. This is especially important if your setup is based on a Virtual Machine: virtual NICs have often unpredictable behavior which depends also on the hypervisor in use; hence, tricks such as putting the virtual NIC in *promiscuous mode* may not work in this case, forcing the user to set the proper MAC address on the pcn-router port.

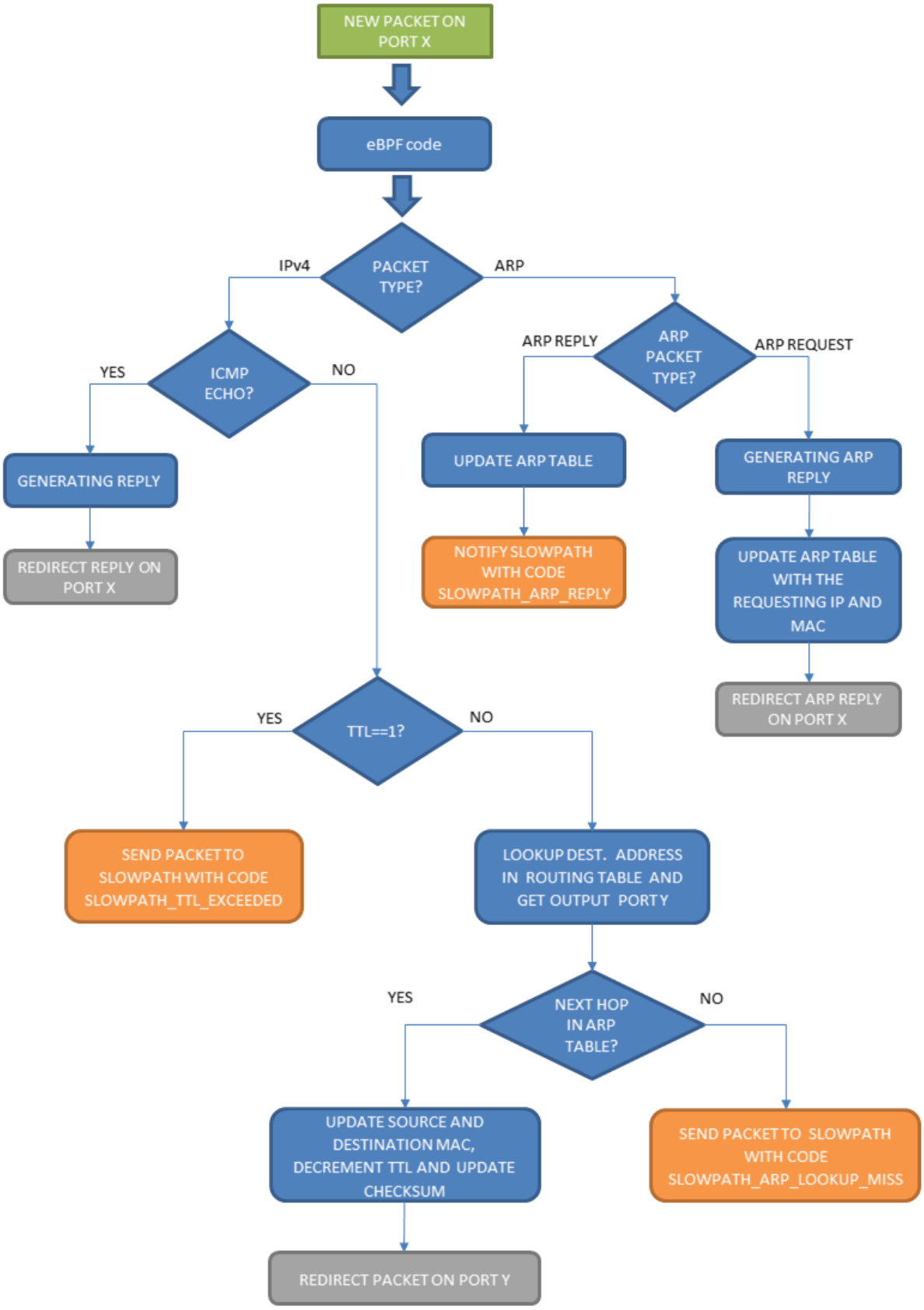
Examples

Please see *Tutorial 2*.

11.9.3 Implementation details

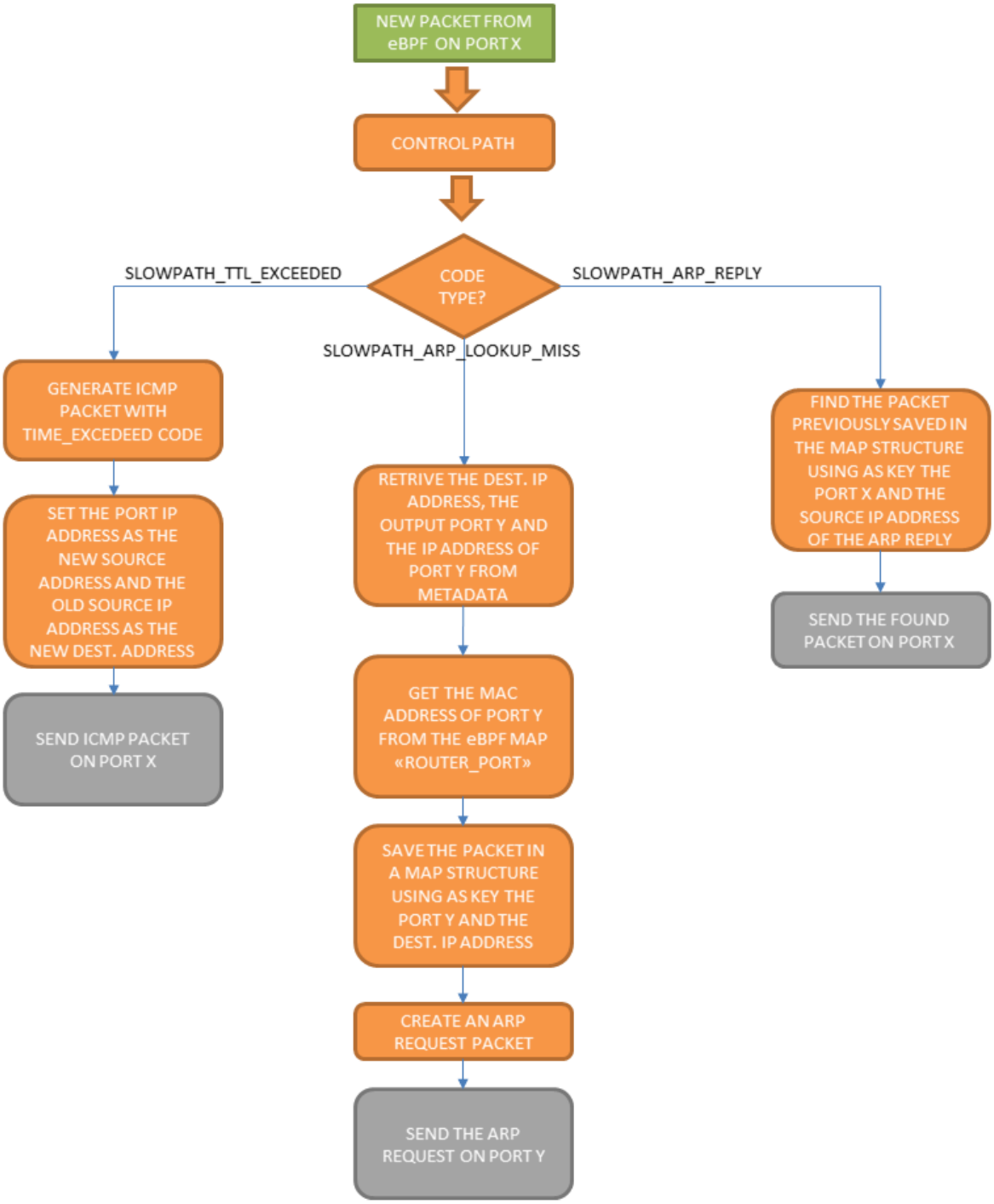
Data plane - fast path

This flowchart summarizes the fast path algorithm of the router, which is implemented in Router_dp.c.


NEW PACKET ON PORT X
eBPF code
PACKET TYPE?
IPv4
ARP
ICMP ECHO?
YES
NO
GENERATING REPLY
REDIRECT REPLY ON PORT X
ARP PACKET TYPE?
ARP REPLY
ARP REQUEST
UPDATE ARP TABLE
NOTIFY SLOWPATH WITH CODE SLOWPATH_ARP_REPLY
GENERATING ARP REPLY
UPDATE ARP TABLE WITH THE REQUESTING IP AND MAC
REDIRECT ARP REPLY ON PORT X
TTL==1?
YES
NO
SEND PACKET TO SLOWPATH WITH CODE SLOWPATH_TTL_EXCEEDED
LOOKUP DEST. ADDRESS IN ROUTING TABLE AND GET OUTPUT PORT Y
NEXT HOP IN ARP TABLE?
YES
NO
UPDATE SOURCE AND DESTINATION MAC, DECREMENT TTL AND UPDATE CHECKSUM
REDIRECT PACKET ON PORT Y
SEND PACKET TO SLOWPATH WITH CODE SLOWPATH_ARP_LOOKUP_MISS

Data plane - slow path

This flowchart summarizes the slow path algorithm of the router, which is implemented in the control plane (source code).


NEW PACKET FROM eBPF ON PORT X
CONTROL PATH
CODE TYPE?
SLOWPATH_TTL_EXCEEDED
SLOWPATH_ARP_LOOKUP_MISS
SLOWPATH_ARP_REPLY
GENERATE ICMP PACKET WITH TIME_EXCEDEED CODE
SET THE PORT IP ADDRESS AS THE NEW SOURCE ADDRESS AND THE OLD SOURCE IP ADDRESS AS THE NEW DEST. ADDRESS
SEND ICMP PACKET ON PORT X
RETRIVE THE DEST. IP ADDRESS, THE OUTPUT PORT Y AND THE IP ADDRESS OF PORT Y FROM METADATA
GET THE MAC ADDRESS OF PORT Y FROM THE eBPF MAP «ROUTER_PORT»
SAVE THE PACKET IN A MAP STRUCTURE USING AS KEY THE PORT Y AND THE DEST. IP ADDRESS
CREATE AN ARP REQUEST PACKET
SEND THE ARP REQUEST ON PORT Y
FIND THE PACKET PREVIOUSLY SAVED IN THE MAP STRUCTURE USING AS KEY THE PORT X AND THE SOURCE IP ADDRESS OF THE ARP REPLY
SEND THE FOUND PACKET ON PORT X

11.10 NAT

This service implements a NAT, network address translation working at layer 3. In particular, the service can remap source and destination IP addresses and source and destination ports in IP datagram packet headers while they are transit across a traffic routing device.

11.10.1 Features

- Transparent NAT
- IPv4 support
- Support ICMP, TCP and UDP traffic
- Support for Source NAT, Masquerade, Destination NAT and Port Forwarding

11.10.2 Limitations

- Unsupported IPv6
- Unsupported ARP: ARP messages are forwarded without any processing
- Incremental external port choice, starting from 1024 each time 65535 is reached

11.10.3 How to use

The NAT is a transparent service, it can be attached to a cube port or to a netdev. The NAT is intended to be used with a router, it may not work when attached to different services.

Attached to a router

When a NAT instance is attached to a router, the external IP is automatically configured.

```
# create nat
polycubectl nat add nat1

# attach to a router port (it should exists)
polycubectl attach nat1 r1:port1
```

Attached to a netdev

```
# create nat
polycubectl nat add nat1

# attach to a router port (it should alraedy exist)
polycubectl attach nat1 eth0

# configure external IP manually
# TODO: there is not an API for it!
```

11.10.4 Working modes

Source NAT

Source NAT selects a given range of hosts (i.e., the `internal-net`) to be translated with a given public IP address (i.e., the `external-ip`). Each time a new session originated from the internal network is recognized, the NAT creates a new binding also for return packets.

Source NAT enables users to specify exactly the range of addresses belonging to the `internal-net`, e.g., excluding some hosts from being natted. In addition, it specifies exactly the external IP that has to be used, e.g., in case multiple external IP addresses are available.

Available and mandatory fields for SNAT rules are:

- `internal-net`: the internal network to translate (e.g 10.0.0.0/24, 10.0.0.1/32)
- `external-ip`: the external IP address (e.g. 1.2.3.4)

Sample rule:

```
polycubectl nat1 rule snat append internal-net=10.0.0.0/24 external-ip=1.2.3.4
```

Masquerade

NAT masquerading is a simpler form of source NAT: it automatically translates the traffic of all hosts coming from the internal network with the IP address currently present on the external interface. Hence, it requires a simpler configuration that is usually enough for most cases.

Unlike `iptables` in Linux, NAT masquerading in Polycube does not add extra overhead compared to SNAT. In fact, the `external-ip` address is kept in a dedicated table in the data plane and it is automatically updated each time the actual IP address configured on the external virtual interface is changed. This avoid to check which IP address has to be used each time a new packet is received.

Masquerade can be either enabled or disabled. When enabled, the source IP address of outgoing packets is set to the external IP of the NAT, and a new source port is assigned.

To enable masquerade:

```
polycubectl nat1 rule masquerade enable
```

To disable masquerade:

```
polycubectl nat1 rule masquerade disable
```

Destination NAT

Destination NAT adds a new translation rule for traffic coming from the external network directed to the internal network. It implements what is also called *static* NAT in other contexts. This features allows external traffic to be forwarded to the internal network without having a corresponding session already established from inside.

The destination NAT forwards all the traffic coming to a specific `external-ip` to a given `internal-ip`, without inspecting other packet fields such as TCP/UDP ports, or protocols (e.g., ICMP). Obviously, this option requires a dedicated `external-ip` for each mapping.

Available and mandatory fields for DNAT rules are:

- `internal-ip`: the internal IP address (e.g 10.0.0.1)

- `external-ip`: the external IP address (e.g. 1.2.3.4)

Sample rule:

```
polycubectl nat1 rule dnat append internal-ip=10.0.0.1 external-ip=1.2.3.4
```

All the packets directed to `1.2.3.4` will be sent to `10.0.0.1`, so use this rule carefully.

Port Forwarding

Port forwarding implements a more selective form of destination NAT, allowing to specify the exact binding by means of other parameters such as the destination TCP/UDP port. This enables the sharing of the same `external-ip` among many `internal-ip` hosts, provided that their traffic can be differentiated e.g., by means of other parameters such as the `external-port` in use.

Available and mandatory fields for Port Forwarding rules are:

- `internal-ip`: the internal IP address (e.g 10.0.0.1)
- `external-ip`: the external IP address (e.g. 1.2.3.4)
- `internal-port`: the internal port number (e.g 8080)
- `external-port`: the external port number (e.g. 80)
- `proto`: the upper layer protocol (e.g. tcp, udp, all). This field is optional: if not specified, all protocols are considered

Sample rule:

```
polycubectl nat1 rule port-forwarding append \
external-ip=1.2.3.4 external-port=80 internal-ip=10.0.0.1 \
internal-port=8080 proto=tcp
```

This type of rule is especially useful to make a server in the inside network reachable from the outside.

11.10.5 Other NAT operations

Deleting rules

It is possible to delete all rules together, all the rules of the same type together, or single rules.

To delete all rules:

```
polycubectl nat1 rule del
```

To delete all rules of a type (SNAT in the example):

```
polycubectl nat1 rule snat del
```

To delete a single rule (an SNAT rule in the example):

```
polycubectl nat1 rule snat entry del RULE_ID
```

Deleting a rule does not affect ongoing natting sessions: to prevent a deleted rule from being applied, flush the natting table.

Showing rules

It is possible to display all existing rules at the same time, or only a specific type of rule.

To display all rules:

```
polycubectl nat1 rule show
```

To display one type of rule (SNAT in the example):

```
polycubectl nat1 rule snat show
```

Rule priority

Explicit rule priorities are not supported; however:

- SNAT rules have higher priority than Masquerade
- Port Forwarding rules have higher priority than DNAT rules
- Port Forwarding rules with protocol have higher priority than Port Forwarding rules without protocol

11.10.6 Natting table

The natting table is used to keep track of the ongoing natting sessions.

Show the natting table:

```
polycubectl nat1 natting-table show
```

To flush the natting table:

```
polycubectl nat1 natting-table del
```

Flushing the natting table is only useful when you want to add a more specific rule for an already active natting session. After the natting table is flushed, the rule with highest priority is applied.

11.10.7 Examples

Some running examples for various configurations can be found in `./test/examples`.

11.11 Load Balancer (DSR)

This service implements a `Direct Server Return-based` Load Balancer working at data-link layer. In this case, Ethernet/IP packets are delivered to the load balancer using its MAC address as destination, which is then replaced with the MAC address of the target server.

11.11.1 Features

- IPv4 support
- Support load balancing of TCP, UDP and ICMP traffic
- Support for a single Virtual IP (VIP)
- Support for ARP reply directed to the VIP address
- Support for two load balancing algorithm(s): *hash on src_ip* or *hash on session parameters* (sip, dip, sport, dport, proto)
- Support for session persistency across multiple servers, through a session table that keeps track of existing connections
- Replaces original destination MAC address with the MAC address of the actual server

11.11.2 Limitations

- Cannot support multiple VIPs
- Does not implement any keep alive mechanism to recognize a server failure
- Servers cannot be removed at runtime

11.11.3 How to use

Configure backend servers

You need to set the VIP on a loopback interface of the server and disable ARP responder (only on loopback interface):

```
sudo ifconfig lo 10.0.0.100 netmask 255.255.255.255 up
sudo sysctl -w net.ipv4.conf.all.arp_ignore=1
sudo sysctl -w net.ipv4.conf.all.arp_announce=2
```

Use the load balancer

```
# create loadbalancer
polycubectl lbdsr add lb1

# add and connect ports
polycubectl lbdsr lb1 ports add frontend-port type=FRONTEND
polycubectl lbdsr lb1 ports add backend-port type=BACKEND
polycubectl lbdsr lb1 ports frontend-port set peer="veth1"
polycubectl lbdsr lb1 ports backend-port set peer="veth2"

# setup frontend vip and mac addresses
polycubectl lbdsr lb1 frontend set mac=aa:bb:cc:dd:ee:ff
polycubectl lbdsr lb1 frontend set vip=10.0.0.100

# configure backend server pool
polycubectl lbdsr lb1 backend pool add 1 mac=01:01:01:01:01:01
```

(continues on next page)

(continued from previous page)

```
polycubectl lbdsr lb1 backend pool add 2 mac=02:02:02:02:02:02
polycubectl lbdsr lb1 backend pool add 3 mac=03:03:03:03:03:03
```

11.12 Load Balancer (RP)

This service implements a `Reverse Proxy Load Balancer`. According to the algorithm, incoming IP packets are delivered to the real servers by replacing their IP destination address with the one of the real server, chosen by the load balancing logic. Hence, IP address rewriting is performed in both directions, for traffic coming from the Internet and the reverse. Packet are hashed to determine which is the correct backend; the hashing function guarantees that all packets belonging to the same TCP/UDP session will always be terminated to the same backend server.

Unknown packets (e.g., ARP; IPv6) are simply forwarded as they are.

11.12.1 Features

- Support for multiple virtual services (with multiple `vip:protocol:port` tuples)
- Support for ICMP Echo Request, hence enabling to `ping` virtual servers
- Session affinity: a TCP session is always terminated to the same backend server even in case the number of backend servers changes at run-time (e.g., a new backend is added)
- Each virtual service can be mapped to a different number of backend servers
- Mapping can be differentiated per protocol (e.g., some backend are dedicated to serve TCP traffic, other UDP, etc)
- Traffic is forwarded to backend services after performing an IP address rewriting in the packet (from `vip:port` to the selected `realip:port`); hence, the `vip` virtual IP address and the IP address of the actual servers should belong to different IP networks
- Support for weighted backends (more later)

11.12.2 Limitations

- Supports only two interfaces

11.12.3 How to use

Weighted backends

Each backend supports a `weight` that determines how incoming sessions are distributed across backends; for example, a backend with `weight = 10` will receive (in average) twice the sessions of another backend with `weight = 5`.

11.12.4 Deployment

A set of `virtual services`, which are specified by a Virtual IP address, protocol and a port (`vip:protocol:port`), are mapped to a given set of `backend services`, actually running on multiple real servers.

Hence, this service exports two network interfaces: - Frontend port: connects the LB to the clients that connect to the virtual service, likely running on the public Internet - Backend port: connects the LB to to backend servers

```
                              +---------------+
[outside network]     (port 0) | loadbalancer  | (port 1)     [inside network]
[public internet]  <-----------| reverse proxy | ---------->  [backend services]
                        (vip)  +---------------+
```

11.12.5 Examples

A possible configuration example is available in the `examples/`.

11.13 Packet capture service

The Packet Capture (`packetcapture`) is a transparent service that allows to capture packets flowing through the interface it is attached to, apply filters and obtain capture in `pcap` format. In particular, the service supports either saving captured packets in the local filesystem (e.g., useful in case of massive load) or it can deliver packets to a remote client that stores them in the remote filesystem.

An example of a client that uses the REST api of the `packetcapture` service is available in the packet capture client folder.

11.13.1 Features

- Transparent service, can be attached to any physical/virtual interface or any Polycube service
- Support for filters (i.e., source prefix, destination prefix, source port, destination port, layer 4 protocol, etc.).
- Support partial capture of packets (i.e., snaplen)
- Support *local mode* (store data locally) or *network mode* (send packets to a remote client) operations.

11.13.2 Limitations

- Traffic is returned as is, without any anonimization primitive.

11.13.3 How to use

The packetcapture service is a transparent service, it can be attached either to a netdev or to a cube port.

Create the service

```
#create the packetcapture service
polycubectl packetcapture add mysniffer capture=bidirectional
```

The `capture` attribute indicates the direction of the packets that the service must capture. Four values are allowed:

- capture only incoming packets: **capture=ingress**
- capture only outgoing packets: **capture=egress**
- capture both incoming and outgoing packets: **capture=bidirectional**
- turn packet capture off: **capture=off**

The direction of the captured packets is independent of the operation in *network mode* or *local mode*.

In this example the service named `mysniffer` will work in bidirectional mode.

Attach to a cube port

```
# Attach the service to a cube port
polycubectl attach mysniffer br1:toveth1
```

Now the packetcapture service is attached to the port `toveth1` of the bridge `br1`:

```
                          +-----------+
veth1 ---**mysniffer**--- |   br1     | ------ veth2
                          +-----------+
```

11.13.4 Set a filter

Traffic can be selected by adding filters with a libpcap-compatible syntax:

```
polycubectl <service name> set filter=<string value>
```

Filter can be set in this way:

- if the filter contains only **one word**: write it normally
- if the filter contains **more than one word**: write the string inside *double quotes* (i.e., " ")
- if you want to capture **all traffic**: write the *all* keyword as string specifier:

```
polycubectl <service name> set filter=all
```

- **default filter**: the service captures no packets (the eBPF datapath simply returns ok)

The currently active filter can be viewed using the command **polycubectl mysniffer filter show**. The current `snaplen` can be viewed using the command **polycubectl mysniffer snaplen show**.

For further details of the implementation of the filter look at the *Implementation details* section.

For more details about the filters supported by libpcap (hence, the syntax allowed to specify filters) see the pcap-filter Linux man page.

Examples of possible filters

```
# Example of the source prefix filter
polycubectl mysniffer set filter="ip src 10.0.2.11"

# Example of the source port filter
polycubectl mysniffer set filter="src port 80"

# Example of the layer 4 protocol filter
polycubectl mysniffer set filter=tcp

# Example of the snaplen filter
# In this case we capture only the first 80 bytes of each packet
polycubectl mysniffer set snaplen=80
```

11.13.5 Get the capture dump

When the service is not set in *networkmode*, the dump is by default written in a resilient way in the temporary user folder. The folder where the dump is written can be changed by using the syntax:

```
polycubectl <service name> set dump="<string value>"

# Example of new dump folder
polycubectl mysniffer set dump="/home/user_name/Desktop/capture"
```

The file extension `.pcap` will be added at the end of the file name.

If a file with the same name already exists, it will be overwritten.

The path of the capture file can be shown using the command `polycubectl mysniffer show dump`.

Otherwise, if the service is set in *network mode*, the capture file can be requested through the use of the provided Python client, or queried simply through the service API.

How to use the packetcapture client

```
# Start the client script
python3 client.py <IPv4 address> <local_dump_name>
```

Set network mode

```
# Start sniffer in network mode
polycubectl mysniffer set networkmode=true

# Start sniffer in local model
polycubectl mysniffer set networkmode=false
```

11.13.6 Implementation details

The pipeline to convert into C code the filtering string entered in the packetcapture service is the following:

pcap filter → *libpcap* → **cBPF** → *cbpf2c* → **C code**

More in details, the first step is to obtain the cBPF (assembly) code from the filtering string, using the `libpcap/tcpdump` format. The filtering string is read from `polycubed` REST interface, then it is compiled in cBPF using the `pcap_compile_nopcap()` function that returns a `bpf_program` structure containing a list of `bpf_insn`.

Then, the code creates a `sock_fprog` structure called `cbpf` that contains all the required filter blocks.

The second step (traslation from cBPF to C) starts with the validation of the cBPF code. Function `_cbpf_dump()` is called for each filtering block and it returns a string containing the equivalent C code for that block.

Inside `_cbpf_dump()`, a switch statement creates two variables, `op` (operation) and `fmt` (operand) depending on the type of instruction of the block (e.g.,return, load, store, alu op. etc.); the above variables will be used to generate the final C code.

This ASM-to-C traslator is ispired to a similar project proposed by Cloudflare; however, in Polycube the translator is written in C/C++ (the CLoudfare one is in Go); furthermore, in Polycube the final output of the translator is a C equivalent of the packet filter, while in the latest version of the Cloudfare project, the final outcome of the translation are eBPF assembly instructions.

The C output facilitates any further modification of the code, e.g., with when additional processing steps are needed, although it impacts on the overall filter conversion time as it requires one additional processing pass involving CLANG/LLVM to convert the C code into eBPF assembly.

Example of C code generated

As a example, we list here is the generated C code for the filter `icmp`:

```
L0:  if ((data + 14) > data_end) {
       return RX_DROP;
     }
     a = ntohs(* ((uint16_t *) &data[12]));
L1:  if (a == 0x0800) {
       goto L2;
     } else {
       goto L5;
     }
L2:  if ((data + 24) > data_end) {
       return RX_DROP;
     }
     a = * ((uint8_t *) &data[23]);
L3:  if (a == 0x01) {
       goto L4;
     } else {
       goto L5;
     }
L4:  pcn_pkt_controller(ctx, md, reason);
L5:  return RX_OK;
```

11.14 Dynamic network monitor

Dynmon (`dynmon`) is a transparent service that allows the dynamic injection of eBPF code in the linux kernel, enabling the monitoring of the network traffic and the collection and exportation of custom metrics.

This service exploits the capabilities of Polycube to replace the eBPF code running in the dataplane and the use of eBPF maps to share data between the control plane and the data plane.

11.14.1 Features

- Transparent service, can be attached to any network interface and Polycube services
- Support for the injection of any eBPF code at runtime
- Support for eBPF maps content exportation through the REST interface as metrics
- Support for two different exportation formats: JSON and OpenMetrics
- Support for both INGRESS and EGRESS program injection with custom rules
- Support for shared maps between INGRESS and EGRESS
- Support for atomic eBPF maps content read thanks to an advanced map swap technique
- Support for eBPF maps content deletion when read

11.14.2 Limitations

- The OpenMetrics format does not support complex data structures, hence the maps are exported only if their value type is a simple type (structs and unions are not supported)
- The OpenMetrics Histogram and Summary metrics are not yet supported
- Data extraction is possible only in the following maps (as listed in [MapExtractor.cpp#L287](https://github.com/polycube-network/polycube/blob/master/src/services/pcn-dynmon/src/extractor/MapExtractor.cpp#L287)): - BPF_MAP_TYPE_HASH, BPF_MAP_TYPE_PERCPU_HASH - BPF_MAP_TYPE_LRU_HASH, BPF_MAP_TYPE_LRU_PERCPU_HASH, - BPF_MAP_TYPE_ARRAY, BPF_MAP_TYPE_PERCPU_ARRAY - BPF_MAP_TYPE_QUEUE, BPF_MAP_TYPE_STACK

Furthermore, optimized data extraction (the so called *batch operations*), are supported only by BPF_MAP_TYPE_HASH, BPF_MAP_TYPE_LRU_HASH, BPF_MAP_TYPE_ARRAY.

11.14.3 How to use

Creating the service

```
#create the dynmon service instance
polycubectl dynmon add monitor
```

Configuring the data plane

In order to configure the dataplane of the service, a configuration JSON object must be sent to the control plane; this action cannot be done through the **polycubectl** tool as it does not handle complex inputs.

To send the data plane configuration to the control plane it is necessary to exploit the REST interface of the service, applying a `PUT` request to the `/dataplane-config` endpoint.

Configuration examples can be found in the *examples* directory.

Attaching to a interface

```
# Attach the service to a network interface
polycubectl attach monitor eno0

# Attach the service to a cube port
polycubectl attach monitor br1:toveth1
```

Collecting metrics

To collect the metrics of the service, two endpoints have been defined to enable the two possible exportation formats:

- JSON format

  ```
  polycubectl monitor metrics show
  ```

- OpenMetrics format

  ```
  polycubectl monitor open-metrics show
  ```

This way, the service will collect all define metrics both for Ingress and Egress program, but if you just want to narrow it down to only one type you can type a more specific command like:

```
polycubectl monitor metrics ingress-metrics show
```

For any other endpoint, try the command line suggestions by typing ? after the command you want to dig more about.

11.14.4 Advanced configuration

For every metric you decide to export you can specify some additional parameter to perform advanced operations:

- Swap the map when read
- Empty the map when read

The JSON configuration will like like so (let's focus only on the Ingress path):

```
{
    "ingress-path": {
        "name": "...",
        "code": "...",
        "metric-configs": [
            {
                "name": "...metric_name...",
```

(continues on next page)

(continued from previous page)

```
                "map-name": "...map_name...",
                "extraction-options": {
                     "swap-on-read": true,
                    "empty-on-read": true
                }
            }
        ]
    },
    "egress-path": {}
}
```

The parameter `empty-on-read` simply teaches Dynmon to erase map content when read. Depending on the map type, the content can be whether completely deleted (like in an HASH map) or zero-ed (like in an ARRAY). This is up to the user, to fit its needs in some more complex scenario than a simple counter extraction.

Concerning the `swap-on-read` parameter, please check the sec-code-rewriter section, where everything is detailed. To briefly sum it up, this parameter allows users to declare swappable maps, meaning that their read is performed ATOMICALLY with respect to the DataPlane (which normally would continue to insert/modify values in the map) thanks to these two steps:

- when the code is injected, the CodeRewriter checks for any maps declared with this parameter and optimizes the code, creating dummy parallel maps to be used later on;
- when the user requires a swappable map content, Dynmon alternatively modifies the current map pointer to point to the original/fake one.

This way, the user will still be able to require the metric he declared as he would normally do, and Dynmon will perform that read atomically swapping the maps under the hoods, teaching DataPlane to use the other parallel one.

11.14.5 Dynmon Injector Tool

This tool allows the creation and the manipulation of a *dynmon* cube without using the standard *polycubectl* CLI.

Install

Some dependencies are required for this tool to run:

```
pip install -r requirements.txt
```

Running the tool

```
Usage: `dynmon_injector.py [-h] [-a ADDRESS] [-p PORT] [-v] cube_name peer_interface␣
↪path_to_dataplane`

positional arguments:
cube_name             indicates the name of the cube
peer_interface        indicates the network interface to connect the cube to
path_to_dataplane     indicates the path to the json file which contains the new␣
↪dataplane configuration
                      which contains the new dataplane code and the metadata associated␣
↪to the exported metrics
```

(continues on next page)

(continued from previous page)

```
optional arguments:
-h, --help                          show this help message and exit
-a ADDRESS, --address ADDRESS       set the polycube daemon ip address (default: localhost)
-p PORT, --port PORT                set the polycube daemon port (default: 9000)
-v, --version                       show program's version number and exit
```

Usage examples

```
basic usage:
./dynmon_injector.py monitor_0 eno1 ../examples/packet_counter.json

setting custom ip address and port to contact the polycube daemon:
./dynmon_injector.py -a 10.0.0.1 -p 5840 monitor_0 eno1 ../examples/packet_counter.json
```

This tool creates a new *dynmon* cube with the given configuration and attaches it to the selected interface.

If the monitor already exists, the tool checks if the attached interface is the same used previously; if not, it detaches the cube from the previous interface and attaches it to the new one; then, the selected dataplane is injected.

11.14.6 Dynmon Extractor tool

This tool allows metric extraction from a *dynmon* cube without using the standard *polycubectl* CLI.

Install

Some dependencies are required for this tool to run:

```
pip install -r requirements.txt
```

Running the tool

```
usage: dynmon_extractor.py [-h] [-a ADDRESS] [-p PORT] [-s] [-t {ingress,egress,all}] [-
↪n NAME]
                           cube_name

positional arguments:
  cube_name             indicates the name of the cube

optional arguments:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        set the polycube daemon ip address (default: localhost)
  -p PORT, --port PORT  set the polycube daemon port (default: 9000)
  -s, --save            store the retrieved metrics in a file (default: False)
  -t {ingress,egress,all}, --type {ingress,egress,all}
                        specify the program type ingress/egress to retrieve the metrics␣
↪(default: all)
  -n NAME, --name NAME  set the name of the metric to be retrieved (default: None)
```

Usage examples

```
basic usage:
./dynmon_extractor.py monitor_0

only ingress metrics and save to json:
./dynmon_extractor.py monitor_0 -t ingress -s
```

11.14.7 Code Rewriter

The Code Rewriter is an extremely advanced code optimizator to adapt user dynamically injected code according to the provided configuration. It basically performs some optimization in order to provide all the requested functionalities keeping high performance and reliability. Moreover, it relies on eBPF code patterns that identify a map and its declaration, so the user does not need to code any additional informations other than the configurations for each metric he wants to retrieve.

First of all, the Rewriter could be accessible to anyone, meaning that other services could use it to compile dynamically injected code, but since Dynmon is the only Polycube's entry point for user code by now, you will se its usage limited to the Dynmon service. For future similar services, remember that this rewriter is available.

The code compilation is performed every time new code is injected, both for Ingress and Egress data path, but actually it will optimize the code only when there is at least one map declared as `"swap-on-read"`. Thus, do not expect different behaviour when inserting input without that option.

There are two different type of compilation:

- PROGRAM_INDEX_SWAP
- PROGRAM_RELOAD

PROGRAM_INDEX_SWAP rewrite

The PROGRAM_INDEX_SWAP rewrite type is the best you can get from this rewriter by now. It is extremely sophisticated and not easy at all to understand, since we have tried to take into account as many scenarios as possible. This said, let's analyze it.

During the first phase of this compilation, all the maps declared with the `"swap-on-read"` feature enabled are parsed, checking if their declaration in the code matches one of the following rules:

- the map is declared as _SHARED
- the map is declared as _PUBLIC
- the map is declared as _PINNED
- the map is declared as "extern"

Since those maps are declared as swappable, if any of these rules is matched, then the rewriter declares another dummy map named `MAP_NAME_1` of the same time, which will be used when the code is swapped. Although, in case the map was _PINNED, the user have to be sure that another pinned map named `MAP_NAME_1` is present and created a priori in the filesystem, since this rewriter cannot create a _PINNED map for you. For all these other types, another parallel map is created smoothly.

If a user created a map of such type, then he probably wants to use another previously declared map out or inside Polycube, or he wanted to share this map between Ingress and Egress programs.

If the map did not match one of these rules, then it is left unchanged in the cloned code, meaning that there will be another program-local map with limited scope that will be read alternatively.

The second phase consists is checking all those maps which are not declared as swappable. The rewriter retrieve all those declarations and checks for them to see if it is able to modify it. In fact, during this phase, whenever it encounters a declaration which it is unable to modify, it stops and uses the PROGRAM_RELOAD compilation as fallback, to let everything run as required, even though in an sub-optimal optimized way.

Since those map must not swap, the rewriter tries to declare a map which is shared among the original and cloned program, in order to make the map visible from both of them. For all those maps, these rules are applied:

- if the map is declared as _PINNED or "extern", then it will be left unchanged in the cloned program, since the user is using an extern map which should exists a priori
- if the map is NOT declared using the standard (BPF_TABLE and BPF_QUEUESTACK) helpers, then the compilation stops and the PROGRAM_RELOAD one is used, since the rewriter is not able by now to change such declarations into specific one (eg. from BPF_ARRAY(...) to BPF_TABLE("array"...), too many possibilities and variadic parameters)
- if the map is declared as _SHARED or _PUBLIC, then the declaration is changed in the cloned code into "extern", meaning that the map is already declared in the original code
- otherwise, the declaration in the original code is changed into BPF_TABLE_SHARED/BPF_QUEUESTACK_SHARED and in the cloned code the map will be declared as "extern". Moreover, the map name will be changed into `MAP_NAME_INGRESS` or `MAP_NAME_EGRESS` to avoid such much to collide with others declared in a different program type.

Once finished, both the original and cloned code are ready to be inserted in the probe. Since it is an enhanced compilation which allows users to save time every time they want to read their metrics, we have used a very efficient technique to alternate the code execution. These two programs are compiled also from LLVM one time, and then they are inserted in the probe but not as primarly code. In fact, this compilation delivers also a master PIVOTING code which will be injected as code to be executed every time there is an incoming/outgoing packet.

The PIVOTING code simply calls the original/cloned program main function according to the current program index. This program index is stored in an internal BPF_TABLE and it is changed every time a user performs a read. When the index refers to the original code, the PIVOTING function will call the original code main function, and vice versa.

Thanks to this technique, every time a user requires metrics there's only almost 4ms overhead due to changing the index from ControlPlane, which compared to the 400ms using the PROGRAM_RELOAD compilation, is an extremely advantage we are proud of having developed.

PROGRAM_RELOAD compilation

This compilation type is quite simple to understand. It is used as a fallback compilation, since it achieves the map swap function, but in a more time expensive way. In fact, when this option is used, it is generated a new code starting from the original injected one, and then the following steps are followed:

1. in the cloned code, change all `MAP_NAME` occurrences with opportunistic names to distinguish them, like `MAP_NAME_1`
2. in the cloned code, add the original `MAP_NAME` declaration that is present in the original code
3. in the original code, add the `MAP_NAME_1` declaration that is present in the cloned code

Since we have to guarantee map read atomicity, we declare a new parallel map with a dummy name. Whenever the user requires metrics, the currently active code is swapped with inactive one, meaning that all the map usages are "replaced" with the dummy/original ones (eg. MAP.lookup() will become MAP_1.lookup() alternatively). Whenever the code is swapped, all the other maps which were not declared as swappable are kept, thanks to the advanced Polycube's map-stealing feature. This way their content is preserved, and the only thing that changes is, as required by the user, the swappable maps' ones.

Both the new and old map declaration need to be places in the codes, otherwise they would not know about the other maps other than the ones they have declared.

The codes are, as said, alternatively injected in the probe, but it is worth noticing that although the PROGRAM_INDEX_SWAP compilation, this one requires LLVM to compile the current code every time it is swapped.

Some tests have been run and their results led to 400ms on average of overhead each time the user requires metrics, due to the LLVM compilation time and the time to inject the code in the probe. Obviously, it is not the better solution, but at least it provides the user all the functionality he asked for, even though the enhanced compilation went wrong.

11.15 Iptables

This cube implements all the control/data plane of `pcn-iptables`. The `pcn-iptables` frontend connects to this service through the `polycubed` REST interface.

Please refer to *pcn-iptables* for more information.

11.16 K8sfilter

`k8sfilter` is a small service that is attached to the physical interface of the nodes and performs a filtering on the incoming packets, if those packets are directed to `NodePort` services they are sent to the `pcn-k8switch`, otherwise packets continue their journey to the Linux networking stack.

This service is not intended to be used alone but with *pcn-k8s*.

11.17 K8switch

This service provides a pod network solution for `kubernetes`, it forwards packets between pods and provides support for `ClusterIP` and `NodePort` services.

Please see *pcn-k8s* to get more information about how to use our kubernetes pod networking solution.

CHAPTER

TWELVE

TUTORIALS

These are a set of step by step tutorials, which show how to interact with `polycube`, and how to deploy simple/complex services, i.e., how to create different network topologies (e.g., service chains).

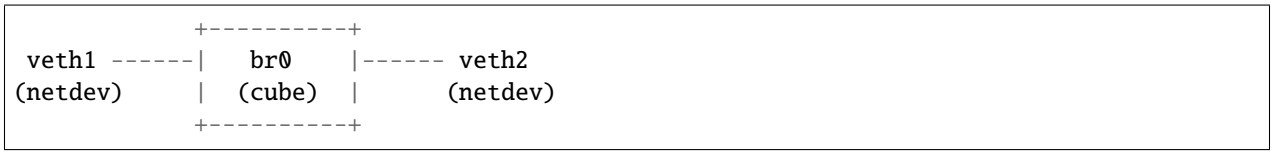
12.1 Prerequisites

Before starting the tutorials, `polycubed` has to be running and `polycubectl` must be available. Please refer to the *Quickstart* document to get those components ready.

12.2 Available tutorials

12.2.1 Tutorial 1: the simplebridge service

This tutorial shows how to create a simple topology with a bridge and a couple of virtual interfaces (*netdev*), belonging to different namespaces, attached to it.



Set up namespaces

The following code configures the network namespaces.

```
# namespace ns1 -> veth1 10.0.0.1/24
# namespace ns2 -> veth2 10.0.0.2/24

for i in `seq 1 2`;
do
    sudo ip netns del ns${i} > /dev/null 2>&1 # remove ns if already existed
    sudo ip link del veth${i} > /dev/null 2>&1

    sudo ip netns add ns${i}
    sudo ip link add veth${i}_ type veth peer name veth${i}
    sudo ip link set veth${i}_ netns ns${i}
    sudo ip netns exec ns${i} ip link set dev veth${i}_ up
```

(continues on next page)

(continued from previous page)

```
    sudo ip link set dev veth${i} up
    sudo ip netns exec ns${i} ifconfig veth${i}_ 10.0.0.${i}/24
done
```

Deploy topology

Step 1: Create the bridge *br1*

```
polycubectl simplebridge add br1
```

If everything goes fine, you shouldn't see any error message on the terminal.

Step 2: Add and connect ports

There are different ways to connect ports to netdevs, this step shows how to use the `connect` command and how to set the `peer` property.

Connect `veth1` **to** `br1`

```
# way 1
# create new port on br1
polycubectl br1 ports add toveth1

# connect port to netdev
polycubectl connect br1:toveth1 veth1
```

Connect `veth2` **to** `br1`

```
# way 2
# create a port and connect to a netdev using a single command
polycubectl br1 ports add toveth2 peer=veth2
```

Show the current status of br1 to check that it is configured as desired (ports are present and its `peer` are the veth interfaces)

```
polycubectl br1 show
name: br1
uuid: ed3e477a-4138-4638-9fe0-7cbd31f1d1fb
type: type_tc
loglevel: info
fdb:
 aging-time: 300

ports:
 name     uuid                                 status  peer   mac
 toveth2  a13d533f-6aeb-49d0-a05c-a048dca31473  up      veth2  1a:68:52:5e:0b:05
 toveth1  d2f1b70b-ff41-4b08-b711-f1b1e9c84786  up      veth1  12:0d:1f:02:d6:3f
```

Step 3: Test the connectivity

Now you can test the connectivity between namespaces using *ping*:

```
# ping ns2 from ns1
sudo ip netns exec ns1 ping 10.0.0.2 -c 1

# ping ns1 from ns2
sudo ip netns exec ns2 ping 10.0.0.1 -c 1
```

After you perform the ping, you can show once again the br1 status, notice that there are new entries in the filtering database.

```
polycubectl br1 show
name: br1
uuid: ed3e477a-4138-4638-9fe0-7cbd31f1d1fb
type: type_tc
loglevel: info
fdb:
 aging-time: 300

 entry:
  address            port     age
  46:2b:1f:a3:d1:81  toveth2  4
  2a:d4:9a:9a:8b:b5  toveth1  4

ports:
 name     uuid                                 status  peer   mac
 toveth2  a13d533f-6aeb-49d0-a05c-a048dca31473  up      veth2  1a:68:52:5e:0b:05
 toveth1  d2f1b70b-ff41-4b08-b711-f1b1e9c84786  up      veth1  12:0d:1f:02:d6:3f
```

Step 4: Remove `br1`

```
# first remove the ports (this step is optional, removing the bridge will do it)
polycubectl br1 del ports toveth1
polycubectl br1 del ports toveth2
# then destroy the bridge
polycubectl del br1
```

12.2.2 Tutorial 2: the router service

This tutorial shows the basic commands of the `router` service. It creates a network topology with one router with three virtual interfaces (simulating three hosts) connected to it. Each host sends packets to its default gateway (i.e., the router interface connected to it), which forwards them to the proper interface based on the (static) routing table.

An introduction to the use of the `polycubectl` is also presented.

For more details about the features of the Router service, please refer to its *description*.

```
             +----------+
 veth1 ------|    r1    |------- veth2
```

(continues on next page)

(continued from previous page)

```
(netdev)     |  (cube)  |       (netdev)
             +----------+
                   |
                   |
                veth3
               (netdev)
```

Set up namespaces

```
# namespace ns1
#  veth1 10.0.1.1/24
#  default gateway 10.0.1.254
# namespace ns2
#  veth2 10.0.2.1/24
#  default gateway 10.0.2.254
# namespace ns3
#  veth3 10.0.3.1/24
#  default gateway 10.0.3.254

for i in `seq 1 3`;
do
    sudo ip netns del ns${i} > /dev/null 2>&1 # remove ns if already existed
    sudo ip link del veth${i} > /dev/null 2>&1

    sudo ip netns add ns${i}
    sudo ip link add veth${i}_ type veth peer name veth${i}
    sudo ip link set veth${i}_ netns ns${i}
    sudo ip netns exec ns${i} ip link set dev veth${i}_ up
    sudo ip link set dev veth${i} up
    sudo ip netns exec ns${i} ifconfig veth${i}_ 10.0.${i}.1/24
    sudo ip netns exec ns${i} route add default gw 10.0.${i}.254 veth${i}_
done
```

Deploy topology

Step 1: Create the router `r1`

```
polycubectl router add r1 loglevel=INFO
```

Step 2: Add and connect ports

Before trying to add a port, let's use the `polycubectl` help to get the info about what are the parameters supported by the port.

```
polycubectl r1 ports add ?

Keyword             Type     Description
 <name>              string   Port Name

 Other parameters:
 peer=value          string   Peer name, such as a network interfaces (e.g., 'veth0') or␣
↪another cube (e.g., 'br1:port2')
 ip=value            string   IP address and prefix of the port
 mac=value           string   MAC address of the port
Example:
 polycubectl r1 ports add port1 peer=r0:port1 ip=207.46.130.1/24 mac=B3:23:45:F5:3A
```

The output indicates that a port name is expected `Keyword`, and following it the `peer`, `ip` and `mac` are supported parameters.

Connect `veth1` to `r1`

```
# create new port on r1 and set the IP parameters
polycubectl r1 ports add to_veth1 ip=10.0.1.254/24
# connect port to netdev
polycubectl connect r1:to_veth1 veth1
```

The router automatically adds a new local entry in the routing table.

```
polycubectl r1 show route
interface  network      nexthop  pathcost
to_veth1   10.0.1.0/24  local    0
```

Connect `veth2` to `r1`

```
# create new port on r1 and set the IP parameters
polycubectl r1 ports add to_veth2 ip=10.0.2.254/24

# connect router port to netdev interface
polycubectl r1 ports to_veth2 set peer=veth2
# You could also used the 'connect' command:
# polycubectl connect r1:to_veth2 veth2
```

Connect `veth3` to `r1`

```
# create new port on r1 and set the IP parameters
# notice that in this case 'peer' is also set so the port is also connected to the netdev
polycubectl r1 ports add to_veth3 ip=10.0.3.254/24 peer=veth3
```

Step 3: Check configuration

You can use the show command to print the whole configuration of the router. You should see an output similar to the next one, where ports are `up` and have `peer` set to the right interface.

```
polycubectl r1 show
name: r1
uuid: b8fd2a02-064e-461e-98d4-d9b7fba384a2
type: type_tc
loglevel: info

ports:
 name      uuid                                  status  peer   ip             mac
 to_veth3  c51bb0ed-9e6f-44ed-a096-b13bc1011331  up      veth3  10.0.3.254/24 ␣
↪72:59:a8:c2:c2:44
 to_veth2  48f8d130-aa32-4354-a1b5-105df9a8ad7b  up      veth2  10.0.2.254/24 ␣
↪d6:42:7f:65:b4:40
 to_veth1  46c685b9-4c80-4466-9d81-985598a07444  up      veth1  10.0.1.254/24 ␣
↪52:f0:5f:2c:a5:a7

route:
 network      nexthop  interface  pathcost
 10.0.1.0/24  local    to_veth1   0
 10.0.2.0/24  local    to_veth2   0
 10.0.3.0/24  local    to_veth3   0
```

Step 4: Test the connectivity between the namespaces and the router

You can test the connectivity between each host (i.e., veth in the namespace) and the router, on all its interfaces, using *ping*:

```
# Ping interfaces from ns1
sudo ip netns exec ns1 ping 10.0.1.254 -c 1
sudo ip netns exec ns1 ping 10.0.2.254 -c 1
sudo ip netns exec ns1 ping 10.0.3.254 -c 1

# Ping interfaces from ns2
sudo ip netns exec ns2 ping 10.0.1.254 -c 1
sudo ip netns exec ns2 ping 10.0.2.254 -c 1
sudo ip netns exec ns2 ping 10.0.3.254 -c 1

# Ping interfaces from ns3
sudo ip netns exec ns3 ping 10.0.1.254 -c 1
sudo ip netns exec ns3 ping 10.0.2.254 -c 1
sudo ip netns exec ns3 ping 10.0.3.254 -c 1
```

Step 5: Test the connectivity between different namespaces

Now you can test the connectivity between all the different namespaces using *ping*:

```
# Ping ns2 from ns1
sudo ip netns exec ns1 ping 10.0.2.1 -c 1

# Ping ns3 from ns1
sudo ip netns exec ns1 ping 10.0.3.1 -c 1

# Ping ns1 from ns2
sudo ip netns exec ns2 ping 10.0.1.1 -c 1

# Ping ns3 from ns2
sudo ip netns exec ns2 ping 10.0.3.1 -c 1

# Ping ns1 from ns3
sudo ip netns exec ns3 ping 10.0.1.1 -c 1

# Ping ns2 from ns3
sudo ip netns exec ns3 ping 10.0.2.1 -c 1
```

Step 6: Remove the router instance

```
# delete r1 and its ports
polycubectl del r1
```

12.2.3 Tutorial 3: creating a service chain with bridges and routers

This tutorial shows how to create a complex service by means of a topology that includes two routers, two bridges and five virtual interfaces (simulating five hosts). Each host is a part of a different network and it sends packets to its default gateway, which forwards them to the proper interface based on the (static) routing table. In this tutorial, the interfaces of the router must have one or more secondary IP addresses to allow hosts belonging to different IP networks to be connected on the same interface of the router.

In addition, this tutorial shows also some more advanced commands of the Router service and how to connect cubes together.

For more details about the features of the Router service, please refer to its *description*.

```
   veth1                                                                  veth3
  (netdev)                                                               (netdev)
10.0.1.1/24                                                           10.0.3.1/24
     |                              10.1.0.2/30                           |
     |                                    |                               |
     |                                    |                               |
+----------+        +----------+          v   +----------+        +----------+
|   br1    |--------|    r1    |--------------|    r2    |--------|   br2    |---------
↪veth5
|  (cube)  |     ^  |  (cube)  |   ^          |  (cube)  |  ^     |  (cube)  |       ␣
↪(netdev)
```

(continues on next page)

(continued from previous page)

```
+---------.+     |  +----------+   |         +----------+  |     +----------+      10.0.
↪5.1/24
     |           |                 |                      |          |
     |          10.0.1.254/24    10.1.0.1/30     10.0.3.254/24        |
     |          10.0.2.254/24                    10.0.4.254/24        |
   veth2                                        10.0.5.254/24       veth4
  (netdev)                                                          (netdev)
10.0.2.1/24                                                       10.0.4.1/24
```

Set up namespaces

```
# Namespace ns1
#  veth1 10.0.1.1/24
#  default gateway 10.0.1.254
# Namespace ns2
#  veth2 10.0.2.1/24
#  default gateway 10.0.2.254
# Namespace ns3
#  veth3 10.0.3.1/24
#  default gateway 10.0.3.254
# Namespace ns4
#  veth4 10.0.4.1/24
#  default gateway 10.0.4.254
# Namespace ns5
#  veth5 10.0.5.1/24
#  default gateway 10.0.5.254

for i in `seq 1 5`;
do
    sudo ip netns del ns${i} > /dev/null 2>&1 # remove ns if already existed
    sudo ip link del veth${i} > /dev/null 2>&1

    sudo ip netns add ns${i}
    sudo ip link add veth${i}_ type veth peer name veth${i}
    sudo ip link set veth${i}_ netns ns${i}
    sudo ip netns exec ns${i} ip link set dev veth${i}_ up
    sudo ip link set dev veth${i} up
    sudo ip netns exec ns${i} ifconfig veth${i}_ 10.0.${i}.1/24
    sudo ip netns exec ns${i} route add default gw 10.0.${i}.254 veth${i}_
done
```

Deploy topology

Step 1: Create cubes, routers r1 and r2, bridges br1 and br2

```
polycubectl router add r1
polycubectl router add r2
polycubectl simplebridge add br1
polycubectl simplebridge add br2
```

Step 2: Add and connect ports on bridges

```
# Connect veth1 and veth2 to br1
polycubectl br1 ports add to_veth1 peer=veth1
polycubectl br1 ports add to_veth2 peer=veth2

# Connect veth3, veth4 and veth5 to br2
polycubectl br2 ports add to_veth3 peer=veth3
polycubectl br2 ports add to_veth4 peer=veth4
polycubectl br2 ports add to_veth5 peer=veth5
```

Step 3: Add and connect ports on routers

Connect `r1` to `br1`

```
# Create new port on r1 and set the IP parameters
polycubectl r1 ports add to_br1 ip=10.0.1.254/24
# Create a bridge interface on br1
polycubectl br1 ports add to_r1
# Connect both ports
polycubectl connect br1:to_r1 r1:to_br1
```

Create secondary address

In order to permit more networks on the same router interface we need to add a secondary address to the router interface which will be the default gateway for a new network. The router automatically adds a local route as before.

```
# Add a secondary address on r1 interface `to_br1`
polycubectl r1 ports to_br1 secondaryip add 10.0.2.254/24
```

Connect `r2` to `br2`

```
# Create new port on r2 and set the IP parameters
polycubectl r2 ports add to_br2 ip=10.0.3.254/24

# Create a bridge interface on br2
polycubectl br2 ports add to_r2
# Connect both ports
polycubectl connect br2:to_r2 r2:to_br2
```

Create secondary address

In the router `r2` we have three different networks, so we need to add two secondary addresses

```
# Add the secondary addresses on r2 interface `to_br2`
polycubectl r2 ports to_br2 secondaryip add 10.0.4.254/24
polycubectl r2 ports to_br2 secondaryip add 10.0.5.254/24
```

Connect the routers

We need to create a point-to-point link between the routers to connect them. To do this, we will use a /30 network

```
# Create new port on r1 and r2 and set the IP parameters
polycubectl r1 ports add to_r2 ip=10.1.0.1/30
polycubectl r2 ports add to_r1 ip=10.1.0.2/30

# Connects the routers
polycubectl connect r1:to_r2 r2:to_r1
```

Step 4: Fill up routing tables

Add static entries in the routing table of router `r1`

We need to tell the router *r1* which are the networks reachable through *r2*

```
polycubectl r1 route add 10.0.3.0/24 10.1.0.2
polycubectl r1 route add 10.0.4.0/24 10.1.0.2
polycubectl r1 route add 10.0.5.0/24 10.1.0.2
```

Add static entries in the routing table of router `r2`

We need to do the same on the router *r2*

```
polycubectl r2 route add 10.0.1.0/24 10.1.0.1
polycubectl r2 route add 10.0.2.0/24 10.1.0.1
```

Show the routing tables of the routers

We can see all the entries of a routing table in a router using the `show` command in the `polycubectl`

```
polycubectl r1 route show
polycubectl r2 route show
```

Step 5: Test the connectivity between the namespaces and the router

You can test the connectivity between each host (i.e., veth in the namespace) and the routers, on all their interfaces, using `ping`:

```
# Ping interfaces from ns1
sudo ip netns exec ns1 ping 10.0.1.254 -c 1
sudo ip netns exec ns1 ping 10.0.2.254 -c 1
sudo ip netns exec ns1 ping 10.1.0.1 -c 1
sudo ip netns exec ns1 ping 10.1.0.2 -c 1

# Ping interfaces from ns2
sudo ip netns exec ns2 ping 10.0.1.254 -c 1
sudo ip netns exec ns2 ping 10.0.2.254 -c 1
```

(continues on next page)

(continued from previous page)

```
sudo ip netns exec ns2 ping 10.1.0.1 -c 1
sudo ip netns exec ns2 ping 10.1.0.2 -c 1

# Ping interfaces from ns3
sudo ip netns exec ns3 ping 10.0.1.254 -c 1
sudo ip netns exec ns3 ping 10.0.2.254 -c 1
sudo ip netns exec ns3 ping 10.1.0.1 -c 1
sudo ip netns exec ns3 ping 10.1.0.2 -c 1

# Ping interfaces from ns4
sudo ip netns exec ns4 ping 10.0.1.254 -c 1
sudo ip netns exec ns4 ping 10.0.2.254 -c 1
sudo ip netns exec ns4 ping 10.1.0.1 -c 1
sudo ip netns exec ns4 ping 10.1.0.2 -c 1

# Ping interfaces from ns5
sudo ip netns exec ns5 ping 10.0.1.254 -c 1
sudo ip netns exec ns5 ping 10.0.2.254 -c 1
sudo ip netns exec ns5 ping 10.1.0.1 -c 1
sudo ip netns exec ns5 ping 10.1.0.2 -c 1
```

You can ping the secondary addresses too.

Step 5: Test the connectivity between different namespaces

Now you can test the connectivity between all the different namespaces:

```
# Ping ns2 from ns1
sudo ip netns exec ns1 ping 10.0.2.1 -c 1

# Ping ns3 from ns1
sudo ip netns exec ns1 ping 10.0.3.1 -c 1

# Ping ns4 from ns1
sudo ip netns exec ns1 ping 10.0.4.1 -c 1

# Ping ns5 from ns1
sudo ip netns exec ns1 ping 10.0.5.1 -c 1
```

and so on.

You can do a complete ping test running ping.sh.

Step 6: Test the *TIME_TO_LIVE EXCEEDED* message

When a router receives a packet with the time-to-live equal or less than 1, it discards the packet and sends to the sender an ICMP TIME_TO_LIVE EXCEEDED message. To test this function you can simply do a ping to an host setting the ttl option to 1.

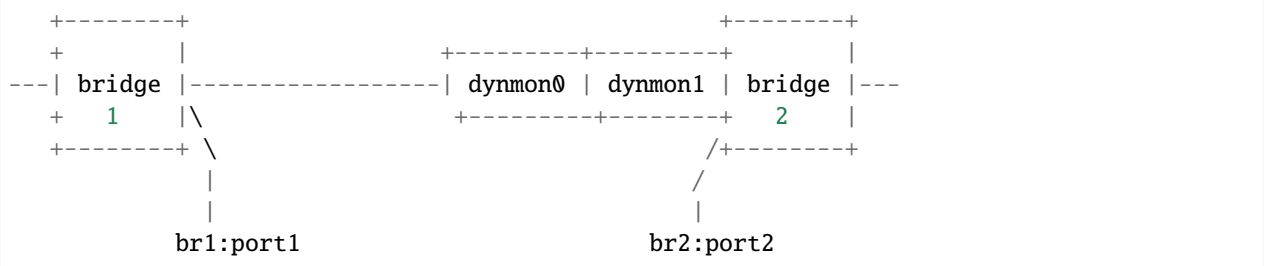
```
# Ping ns2 from ns1 with TTL=1
sudo ip netns exec ns1 ping 10.0.2.1 -c 1 -t 1
```

12.2.4 Tutorial 4: Attach multiple transparent cubes

This simple tutorial aims to show how multiple cubes can be attached to the same interface/port. For the sake of simplicity the configuration presented in this tutorial is really simple, but enough to understand the principle behind these operations.

Before to start, remind that not only different transparent cubes (such as firewall, nat, etc.) can be attached in pipeline to the same interface, but also cubes of the same type can be inserted, like two firewalls.

The transparent cubes used as example in this tutorial are two different instances of *description*, which will be attached to a *description* port as in figure.



Set up bridges

To set up the two bridges, please refer to this *description*, which describes how to configure the cubes and create a link between two ports.

Step1: Create transparent cubes

Now that the basic topology is ready, create the two `dynmon` instances:

```
polycubectl dynmon add monitorA
polycubectl dynmon add monitorB
```

Step2: Attach cubes

To completely understand all the possibility the framework offers to attach cubes, the command line helper can be used. In bash (in many shells may not be configured by default), type the following command to see all the possible parameters:

```
~$ polycubectl attach ?

keyword         type                 description
<cube>          cube                 Transparent cube to be attached (E.g. nat1)
<port>          cube:port or netdev  Port of a cube or netdev to attach the transparent␣
↪cube (E.g. rt2:port2 or eth0)
position=value                       Position to place the cube. auto, first, last
before=value                         Place the cube before another one
after=value                          Place the cube after another one

Example:
 attach nat1 rt:port1
```

As in our case, when more than one cube need to be attached in pipeline to the same interface, one parameter between position/before/after should be specified, otherwise the framework cannot understand where the user wants that cube to be placed.

Firstly, to attach *monitor1* to *br2:port2*:

```
~$ polycubectl attach monitor1 br2:port2
```

Notice that since there are not other cubes attached yet, the position can be omitted.

Finally, to attach also *monitor1* to *monitor2*:

```
~$ polycubectl attach monitor2 br2:port2 after=monitor1
#
# OR
#
~$ polycubectl attach monitor2 br2:port2 position=last
```

Now the configuration is completed. Every `dynmon` instance is working standalone and each one can be configured differently.

Last consideration:

Pay attention to the order, especially when different services which can eventually decide to drop packets are working in pipeline, since the final result may not be as expected (if `dynmon0` decided to drop a packet, `dynmon1` would not receive it).

CHAPTER

THIRTEEN

POLYCUBE DEVELOPERS GUIDE

This guide represents an initial starting point for developers that want to implement new services (e.g., custom NAT, router, etc) using the Polycube software infrastructure.

13.1 Writing the eBPF datapath

eBPF is an extension of the traditional Berkeley Packet Filter. The Polycube architecture leverages the software abstraction provided by BCC, which is further extended in this project particular with respect to eBPF features that are useful for networking services. In order to get more information about how to use the maps in BCC please read the BCC reference guide, additionally there is a list of the available eBPF helpers.

Polycube architecture adds a wrapper around the user's code, this wrapper calls the `handle_rx` function with the following parameters:

- **ctx**: packet to be processed
- **md**: packet metadata
- **in_port**: integer that identifies the ingress port of the packet.

Polycube provides a set of functions to handle the packets, the return value of the `handle_rx` function should be the result of calling one of these functions.

- **pcn_pkt_redirect(struct __sk_buff *skb, struct pkt_metadata *md, u16 port);**: sends the packet through an the `ifc` port. Example: Helloworld service.
- **pcn_pkt_drop(struct __sk_buff *skb, struct pkt_metadata *md);**: drops the packet. It is the same that just returning *RX_DROP*. Example: Helloworld service.
- **pcn_pkt_redirect_ns(struct __sk_buff *skb, struct pkt_metadata *md, u16 port)**: (available only for *shadow* services) sends the packet to the namespace if it comes from the port indicated as parameter.

13.1.1 Processing packets in the slowpath

A copy of the packet can be sent to the controller to be processed by the slowpath using the following helpers:

- **pcn_pkt_controller(struct __sk_buff *skb, struct pkt_metadata *md, u16 reason)**: sends a copy of the packet to the controller. Reason can be used to indicate why the packet is being sent to the custom code running in the control path.

Example: Helloworld service.

- **pcn_pkt_controller_with_metadata(struct __sk_buff *skb, struct pkt_metadata *md, u16 reason, u32 metadata[3])**: sends a copy of the packet to the custom code running in the control path. In addition to the reason the user can also send some additional metadata.

The packet will be processed by the `packet_in` method of the controller.

13.1.2 Checksum calculation

The L3 (IP) and L4 (TCP, UDP) checksums has to be updated when fields in the packets are changed. `polycube` provides a set of wrappers of the eBPF helpers to do it:

- **pcn_csum_diff()**: wrapper of BPF_FUNC_csum_diff

Note that in case of XDP cubes and kernels version prior to 4.16 this function supports only 4 bytes arguments.

- **pcn_l3_csum_replace()**: wrapper of BPF_FUNC_l3_csum_replace
- **pcn_l4_csum_replace()**: wrapper of BPF_FUNC_l4_csum_replace

Services as NAT and Load Balancer show how to use these functions.

13.1.3 VLAN Support

The VLAN handling in TC and XDP eBPF programs is a little bit different, so polycube includes a set of helpers to uniform this accross.

- bool **pcn_is_vlan_present** (struct CTXTYPE* pkt)
- int **pcn_get_vlan_id** (struct CTXTYPE* pkt, uint16_t* vlan_id, uint16_t* eth_proto);
- uint8_t **pcn_vlan_pop_tag** (struct CTXTYPE* pkt);
- uint8_t **pcn_vlan_push_tag** (struct CTXTYPE* pkt, u16 eth_proto, u32 vlan_id);

13.1.4 Timestamping

So far we know that computing timestamp in eBPF is quite tricky, due to the lack of usable kernel helpers. The only supported function is **bpf_ktime_get_ns()**, which returns the value of a kernel MONOTONIC clock, meaning that if the device has gone to sleep/suspend that clock is not updated. Moreover, in the **ctx** structure the field **ctx->tstamp** has not a standard behaviour: we tested that this value can be:

- empty
- Unix epoch timestamp
- Unknown (maybe kernel timer) timestamp

We have introduced an helper function to compute such timestamp referring to Unix Epoch: **pcn_get_time_epoch()**. What basically happens is that to all eBPF program is injected a flag **_EPOCH_BASE** containing a precomputed value. This value represents the **Unix_Epoch - Kernel_timer** at time X. Calling the helper would increase this value by **bpf_ktime_get_ns()** nanoseconds, thanks to you obtain the Unix Epoch time of when you call such helper.

In the following release, it will be possible to compute the exact timestamp even though the device sleeps/suspends due to the introduction of a new **bpf_ktime_get_ns()** function, which in addition will consider the inactivity time (CLOCK_BOOTTIME instead of CLOCK_MONOTONIC).

13.1.5 Known limitations

- Since you cannot send a packet on multiple ports, multicast, broadcast or any similar functionality has to be implemented in the control path.
- The support for multiple eBPF programs is not yet documented.
- Timestamp will not be exact if the device running Polycube sleeps/suspends during its execution.

13.1.6 Debugging the data plane

See how to debug by *logging in the dataplane*.

13.2 Writing the control plane

13.2.1 Handling eBPF programs

A cube is composed by 0 or more eBPF programs, the master program is the first that receives the packets to be processed.

The `polycube::service::Cube` and the `polycube::service::TransparentCube` classes provide wrapper to manage eBPF programs, it allows to load, unload and reload them. The constructor of such classes receives two `std::vector<std::string>` that contains the code for the ingress and egress chain programs, those programs will be compiled and loaded when the cube is created, this is possible to create a `Cube` without any program and add them later on.

The following functions allow to reload, add and delete eBPF programs.

- void reload(const std::string &code, int index, ProgramType type)
- int add_program(const std::string &code, int index, ProgramType type)
- void del_program(int index, ProgramType type)

Where:

- index: position of the program
- type: if the program is on the `INGRESS` or `EGRESS` chain.

13.2.2 Adding and removing ports

The main service class has two methods, `addPorts` and `removePorts`, these methods are automatically called by the infrastructure. Within these functions, there must be a call to `Cube::add_port()` and `Cube::remove_port()` respectively, these functions allow the infrastructure to perform all the required work for adding or removing the port.

The `Cube::add_port` function receives a `PortsJsonObject` object with the initial port configuration.

The stub generated already include those functions, so in general the developer does not have to worry about it.

13.2.3 eBPF maps API

The `polycube::service::BaseCube` base class provides the `get_{raw, array, percpuarray, hash, percpuhash}_table()` methods that allows to get a reference to an eBPF map.

The `RawTable` is intended to access the memory of the maps without any formatting, the user should pass pointers to key and/or value buffers. It provides the following API:

- void set(const void* key, const void* value)
- void get(const void* key, void * value)
- void remove(const void* key)

The `ArrayTable` and `PercpuArrayTable` are intended to handle array like maps, this class is templated on the value type.

The `ArrayTable` provides the following API:

- void set(const uint32_t &key, const ValueType &value)
- ValueType get(const uint32_t &key)
- std::vector<std::pair<uint32_t, std::vector<ValueType>>> get_all()

The the `PercpuArrayTable` provides:

- std::vector<ValueType> get(const uint32_t &key)
- std::vector<std::pair<uint32_t, std::vector<ValueType>>> get_all()
- void set(const uint32_t &key, const std::vector<ValueType> &value)
- void set(const uint32_t &key, const ValueType &value): Set value in all CPUs.

The `HashTable` and `PercpuHashTable` are intended to handle hash like maps, this class is templated on the key and value type. The `HashTable` provides the following API:

- ValueType get(const KeyType &key)
- std::vector<std::pair<KeyType, ValueType>> get_all()
- void set(const KeyType &key, const ValueType &value)
- void remove(const KeyType &key)
- void remove_all()

The `PercpuHashTable` exposes:

- std::vector<ValueType> get(const KeyType &key)
- std::vector<std::pair<KeyType, std::vector<ValueType>>> get_all()
- void set(const KeyType &key, const std::vector<ValueType> &value)
- void set(const KeyType &key, const ValueType &value)**: Set value in all CPUs.
- void remove(const KeyType &key)
- void remove_all()

In order to have an idea of how to implement this, take at look at the already implemented services, router and firewall are good examples.

13.2.4 Implementing the control path

Handling PacketIn events

If the service is intended to receive data packets in the control path (using an approach that is commonly said *slow path*), you sh

- **port**: a reference to the ingress port of the packet
- **md**: the metadata associated to this packet
- **reason** and **metadata**: values used by the datapath when sending the packet to the control path through the `pcn_pkt_controller()` functions.
- **packet**: an array containing the packet's bytes.

Generating PacketOut events

The `Port` class contains the `send_packet_out(EthernetII &packet, bool recirculate = false)` method that allows to inject packets into the datapath. The `recirculate` parameter specifies if the packet should be sent out of the port (`recirculate = false`) or received through the port (`recirculate = true`).

Only in shadow services the `Port` class contains the `send_packet_ns(EthernetII &packet)` method that allows to send packets into the service namespace.

A reference to a port can be got using the `get_port()` function of the Cube base class.

13.2.5 Debugging the control plane

See how to debug by *logging in the control plane*.

13.3 Writing a YANG data model

The service structure is described using a YANG data model. All the services intended to work with polycube must derive from a base service definition that defines some common structures across all services (e.g., *ports*).

Polcyube includes three different base datamodels:

- polycube-base: common definitions for all cube types
- polycube-standard-base: base datamodel for standard cubes
- polycube-transparent-base: base datamodel for transparent cubes

The `polycube-base` datamodel must be always present, and `polycube-standard-base` or `polycube-trasparent-base` depending on the type of cube you are developing.

In order to derive from such base datamodel include the following line:

```
import polycube-base { prefix "polycube-base"; }
import polycube-standard-base { prefix "polycube-standard-base"; }
// or
import polycube-base { prefix "polycube-base"; }
import polycube-transparent-base { prefix "polycube-transparent-base"; }
```

In addition, if you are implementing a standard cube and the base definition of a port is not enough for the service, it is possible to use the *augment* keyword to insert additional nodes, as shown below:

```
uses "polycube-standard-base:standard-base-yang-module" {
  augment ports {
  // Put here additional port's fields.
  }
}
```

The easiest way for starting to write a datamodel is to take a look to the existing ones:

- helloworld.yang
- simpleforwarder.yang
- simplebridge.yang

Documentation about YANG can be found on RFC6020 and RFC7950.

13.4 Polycube Automatic Code Generation Tools

13.4.1 Generating a service stub

polycube-codegen is used to generate a stub from a YANG datamodel. It is composed by `pyang`, that parses the YANG datamodel and creates an intermediate JSON that is compliant with the OpenAPI specifications; and `swagger-codegen` starts from the latter file and creates a C++ code skeleton that actually implements the service. Among the services that are automatically provided, the C++ skeleton handles the case in which a complex object is requested, such as the entire configuration of the service, which is returned by disaggregating the big request into a set of smaller requests for each *leaf* object. Hence, this leaves to the programmer only the responsibility to implement the interaction with *leaf* objects.

The easiest way to generate a stub is to use the `polycubenets/polycube-codegen` docker image. This contains `pyang`, `swagger-codegen` and a script to invoke them.

In order to run the image, you need to mount a volume into `/polycube-base-datamodels` where the base datamodels are located on the host (by default `/usr/local/include/polycube/datamodel-common/`). Furthermore, you need also to mount a volume to share the input yang model and the output folder.

```
export POLYCUBE_BASEMODELS=<path to base models usually /usr/local/include/polycube/
↪datamodel-common/>
docker pull polycubenets/polycube-codegen
 docker run -it --user `id -u` \
  -v $POLYCUBE_BASEMODELS:/polycube-base-datamodels \
  -v <input folder in host>:/input \
  -v <output folder in host>:/output \
  polycubenets/polycube-codegen \
  -i /input/<yang datamodel> \
  -o /output/<output folder>
```

For instance, the following command is used to generate the stub for the `pcn-iptables` service.

```
export POLYCUBE_BASEMODELS="/usr/local/include/polycube/datamodel-common/"
docker pull polycubenets/polycube-codegen
docker run -it --user `id -u` \
  -v $POLYCUBE_BASEMODELS:/polycube-base-datamodels \
  -v $HOME/datamodels/input:/input \
  -v $HOME/datamodels/output:/output \
```

(continues on next page)

(continued from previous page)

```
polycubenets/polycube-codegen \
-i /input/iptables.yang \
-o /output/iptables_stub
```

Stub Code Structure

- **src/api/{service-name}Api.[h,cpp]** and **src/api/{service-name}ApiImpl.[h,cpp]**: Implements the shared library entry points to handle the different request for the rest API endpoints. The developers does not have to modify it.
- **src/interface/**: This folder contains one pure virtual class for each object used in the control API. Each class contains the methods that must be implemented in each object class.
- **src/serializer/**: This folder contains one JSON object class for each object used in the control API. These classes are used to performs the marshalling/unmarshalling operations.
- **src/{object-name}.h**: One header file for each object used in the control API. These classes implement the corresponding object interface stored in the *src/interface* folder.
- **src/{object-name}.cpp**: These files contain the getter and setter methods, constructor and destructor and finally some methods to handle the object life cycle such as create, delete, replace, update and read. Some methods contain a basic implementation, others instead have to be implemented.
- **src/{service-name}_dp.c**: Contains the datapath code for the service.
- **src/{service-name]-lib.cpp**: Is used to compile the service as shared library.
- **.swagger-codegen-ignore**: This file is used to prevent files from being overwritten by the generator if a re-generation is performed.

Starting from the generated classes, the developer only has to modify the `src/{object-name}.h`, `src/{object-name}.cpp` and `src/{service-name}_dp.c` files.

The `src/{object-name}.cpp` files contain the stub for the methods that the developer has to modify. Some of them have a basic implementation and the developer is suggested to improve it, others present only the stub. The developer has to implement constructor and destructor for each object. Moreover in these classes the developer has to implement the getter and setter methods for each parameter contained in the object. In order to achieve the target, the developer is free to add fields and methods to these classes and also add other classes if necessary.

Updating an existing service stub

This step is required if the service skeleton already exists and the programmer has to update the YANG model with new features. This procedure avoids to overwrite all the source code already generated, by adopting an incremental approach that minimizes the changes in the existing code to the developer.

If the developer has to re-generate the service stub (because he modified something in datamodel), we suggest to avoid the re-generation of some files by properly modifying the *.swagger-codegen-ignore* file. All the files are considered in the *src/* folder (except for the files in *src/api/* folder). To avoid a file re-generation the developer has to write the file name in the *.swagger-codegen-ignore* file. If the file name is preceded by the *!* character or it is not present, the file will be overwritten. For default the *.swagger-codegen-ignore* avoids the re-generation of all the files that the developer has to modify (`src/{object-name}.h`, `src/{object-name}.cpp` and `src/{service-name}_dp.c` files).

Important: do not remove the line `.swagger-codegen-ignore` present in the `.swagger-codegen-ignore` file because otherwise this file will be overwritten and this causes the loss of previous modifications to the file.

13.5 Debugging Polycube services

The main way to debug a service in Polycube is by using debug log messages. Polycube provides the primitives that enables to print messages (from both data and control plane) in the **same** log file, hence facilitating the integrated debugging of both data and control plane components. However, primitives to be used are different in the above two cases and will be presented later in this document.

By default, the log file can be found in `/var/log/polycube/polycubed.log` and it contains all the output produced by the polycubed daemon and the running services.

13.5.1 Configuring logging levels

Polycube defines six log levels, listed below in increasing order of importance, following the general accepted practice proposed on StackOverflow:

- `trace`: when the developer would "trace" the code and trying to find one part of a function specifically.
- `debug`: information that is diagnostically helpful to people more than just developers (IT, sysadmins, etc.).
- `info` (*default value*): information that is usually useful to log (service start/stop, configuration assumptions, etc); information that users would like to have available but usually do not care about under normal circumstances.
- `warning`: anything that can potentially cause application oddities, but that can usually be automatically recovered by the framework, such as switching from a primary to backup server, retrying an operation, missing secondary data, etc.
- `error`: any error which is fatal to the operation, but not the service or application, such as cannot open a required file, missing data, etc.. These errors should force user (administrator, or direct user) intervention.
- `critical`: any error that is forcing a shutdown of the service or application to prevent data loss (or further data loss). These errors should be reserved only for the most heinous errors and situations where there is guaranteed to have been data corruption or loss.

Each Polycube service has a `loglevel` property that enables the printing of log messages filtered by the choosen level. Setting the `loglevel` to `X` means that all the messages of that loglevel and above will be printed. For instance, if `loglevel=warning`, all messages marked as `warning`, `error` and `critical` are printed.

Setting the `loglevel` property of a service can be achieved by creating the service with the proper property, such as in the following:

```
polycubectl router r0 add loglevel=debug
```

This creates a new instance of a router, called `r0`, with the highest logging level (`debug`) turned on.

Note 1 The loglevel of each individual service does not affect the loglevel of the `polycubectl` command line. In other words, `polycubectl` logging can be set to minimal (i.e., logging level = `info`), while a given service can have the logging level equal to `debug`. More information is available in the *configuring polycubectl* section.

Note 2 The loglevel of each individual service does not affect the loglevel of the `polycubed` system daemon. In other words, `polycubed` logging can be set to minimal (i.e., logging level = `info`), while a given service can have the logging level equal to `debug`. More information is available in the *debugging the polycubed daemon* section.

Note 3: the log file can be easily filtered on either (1) a specific loglevel or (2) a specific service by using the standard Unix `grep` tool.

Usage examples:

```
sudo polycubed | grep "debug"
sudo polycubed | grep -E "warning|error|critical"
sudo polycubed | grep "bridge"
```

13.5.2 Logging the eBPF data plane

The common eBPF primitive `bpf_trace_printk()`, which is widely used in other eBPF frameworks, should not be used in Polycube. Polycube offers a new `pcn_log()` primitive, which integrates both data and control plane logging in the same location.

Syntax:

```
  pcn_log(ctx, level, msg, args...)

- ``ctx``: packet being processed.
- ``level``: type of logging level: ``LOG_TRACE``, ``LOG_DEBUG``, ``LOG_INFO``, ``LOG_
↪WARN``, ``LOG_ERR``, ``LOG_CRITICAL``.
- ``msg``: message to print
- ``args``: arguments of the message
```

Three special format specifiers are available:

- `%I`: print an IP address
- `%M`: print a MAC address
- `%P`: print a TCP/UDP port

Please note the the custom specifiers expect the data to be in *network byte order* while standard specifiers expects it to be in *host byte order*.

Current limitations of `pcn_log()`:

- Cannot be used inside a macro
- Maximum 4 arguments are allowed

Usage example:

```
pcn_log(ctx, LOG_DEBUG, "Receiving packet from port %d", md->in_port);
```

13.5.3 Logging the control plane

Custom `printf()` or similar primitives, which make the code difficult to debug, should not be used in the Polycube control plane. In fact, the `pcn_log()` primitive presented below can be used also in the control plane, as Polycube includes a powerful logging system implemented within a dedicated class.

Usage example:

```
logger()->info("Connected port {0}", port_name);
logger()->debug("Packet size: {0}", packet.size());
```

13.6 Debugging using Integrated Development Environment (IDE)

13.6.1 VS Code

Polycube uses `cmake` to create the compilation environment (Makefiles, etc), which, by default is configured to create the executables in `Release` mode. To debug your code, you must compile Polycube in `Debug` mode, with the following commands (follow also here):

```
mkdir Debug
cd Debug
cmake -DCMAKE_BUILD_TYPE=Debug ..
make -j
```

Once all the Polycube executables have been created with _debug_ information, you must install them in the default folders with the following command:

```
sudo make install
```

Now, let's create a `launch.json` file, following the VS Code instructions:

1. Run
2. Start Debugging (or Add Configuration)
3. Select the C ++ template (GDB/LLDB)
4. Default configuration
5. As "program", add `/usr/local/bin/polycubed`, which represents the default location. Alternatively, if you modified the installation script, enter the path that comes out as a result of the `which polycubed` command
6. As "args", add `["--loglevel=DEBUG"]` to enable Polycube debug mode logging, which provides diagnostic information useful to people more than just developers. Note that the Polycube _debug_ logging (which simply means _the highlest level of **verbosity**_) is independent from the build type, hence it has to be enabled explicitly.
7. Later, also following this link, create a file called "gdb" in "home/{username}" (for example) and enter `pkexec /usr/bin/gdb "$@"` to make sure that you are prompted to enter the password at startup. This is because Polycube requires root permissions to run. Finally just edit the `launch.json` file with these 3 lines:

```
"externalConsole": false,
"miDebuggerPath": "/home/<username>/gdb",
"MIMode": "gdb",
```

The final `launch.json` file should be something like the following:

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "(gdb) Launch",
```

(continues on next page)

(continued from previous page)

```
            "type": "cppdbg",
            "request": "launch",
            "program": "/usr/local/bin/polycubed",
            "args": ["--loglevel=DEBUG"],
            "stopAtEntry": false,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": false,
            "miDebuggerPath": "/home/pino/gdb",
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "Abilita la riformattazione per gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ]
        }
    ]
}
```

13.6.2 CLion

If you use CLion, you can debug Polycube with the following steps:

1. Run CLion with sudo (Polycube needs root permissions)
2. Set breakpoints and build/install Polycube (as explained above)
3. At this point there are two equivalent ways to debug Polycube with CLion:
    a. Use the CLion debugger: at the top right select Debug 'polycube';
    b. (or) Run Polycube in a terminal with enabled debug logs (e.g., `sudo polycubed --loglevel = DEBUG`); then, from CLion, go to Run->Attach to Process.. and search for "polycubed"
4. At this point, from another terminal, just use `polycubectl` to interact with Polycube.

13.7 Debugging network traffic using tcpdump/Wireshark

Debugging (i.e., sniffing) network traffic flowing *between* two Polycube services (e.g., on the link that connects a Polycube router to a Polycube bridge) can be achieved with the *span mode*.

// TODO

13.8 Profiler Framework

Simple, easy to use and fast C++ header class framework for profiling and benchmark.

13.8.1 Introduction

This framework provides an efficient and simple benchmarking primitive for user programs. Particularly, this can be used to profile the control plane portion of Polycube. Although there are several powerful tools (particularly in Linux) that target benchmarking and profiling, they are often difficult to learn for an occasional user, which may need to perform some simple and fast benchmarking.

A worth-mentioning tool is perf. Perf is a Linux profiling tool with performance counters which helps users to understand at low-level how not only the program, but also the hardware behaves. As the wiki reports, this tool not only is quite complex to learn, but it also produces deep information that users may not use or that they have to aggregate them to achieve a readable result. Moreover, perf refers to different types of event (ex. I/O disk) users may not be interested in.

This library provides a simple to use method not only to run a whole program benchmark, but also to measure single code-fragments performance.

13.8.2 How it works

The framework defines two different main components:

- Check-point: `CHECKPOINT(<name>)`
- Store-point: `STOREPOINT(<name>)`

A check-point is a point of the code where we simply want to take a measure (timestamp).

A store-point is a specialized check-point which not only acts like it, but it also manages the data storing into a file.

While the usage of a check-point leads to a negligible overhead (~60 nanoseconds), it is suggested to use wisely the store-points, since the I/O operation on file are more expensive.

Each measure is characterized by:

- filename: the name of the file which contains the check-point;
- line of code: the line of code where the check-point has been inserted;
- name (optional): the name/description of the checkpoint given by the user;
- timestamp: the measure in *nanoseconds* taken.

Accordingly, the format used to store check-points is: <*filename*>,<*line_of_code*>,<*name*>,<*timestamp*>.

The output file would be available at `/var/log/polycube/profile_<date>_<time>).log` where date and time refer to execution moment.

13.8.3 Usage

The framework is turned on by defining `#define PROFILER` in your code (wherever the user wants) but **before** including the profiling header (i.e., *#include <polycube/profile.h>*). Without this definition, the framework will be automatically turned off for all the execution.

Possible operations:

- to insert Check-points, the user have just to insert the macro `CHECKPOINT(<name>)`.
- to insert Store-points, insert the macro `STOREPOINT(<name>)`.
- to remove them, comment the proper line or remove it.

An interesting feature is that in case the user deactivates the framework, it is not required to remove every checkpoints he had previously inserted in the code: the framework will handle it by replacing them with an empty line.

The output file created by the framework can be easily imported by every spreadsheet program (LibreOffice, Excel,...), but since it does not contain clear data, the user can use the script `profiler_parser.py` (Doc. inside the script) to achieve a more complete, elegant and importable as well result.

13.8.4 Example

In this simple program we want to measure the performance of our function to compute the square of a number. We insert two check-points between the functions call and a final store-point at the end.

```
#define PROFILER
#include <polycube/profiler.h>

int main(int argc, char **argv) {
  CHECKPOINT("Beginning")                 //line 7
  computeSquare(1);
  CHECKPOINT("After first computation")   //line 9
  computeSquare(2);
  STOREPOINT("After second computation")  //line 11
  return 0;
}
```

After the execution, we obtain the following `profile_20190902_181847.log` file:

```
main.cpp,7,Beginning,1567333232104876595
main.cpp,9,After first computation,1567333232104876666
main.cpp,11,After second computation,1567333232104876737
```

Since the stored timestamps refer to the start of the epoch, to beautify the data and print some statistics we use our script `profiler_parser.py`:

```
 $ python profiler_parser.py profile_20190902_181847.log


=================================================================================
||FROM                    |TO                        |TIME(ns)            ||
=================================================================================
||Beginning               |After first computation |71                    ||
||After first computation|After second computation|71                    ||
=================================================================================
Max execution time: 71 ns
```

(continues on next page)

(continued from previous page)

```
Min execution time: 71 ns
Avg execution time: 71.0 ns
```

In case some checkpoint has not been given the name, the displayed information will follow the format `<file>,<line>` in order to identify the checkpoint as well. The script has another feature available in case we would like to export data in .csv format: it can detect similar checkpoints inside a loop by comparing all the available information and it assigns them an additional identifier to clarify the differences among them.

13.8.5 Performance analysis

For this analysis, tests have been run on a Dell Xps 9570. Please consider that results are different depending on your processors and the available CPU.

The test has been run on a simple file where we inserted 100 consecutively check-points and a final store-point. The check-points are inserted inside a loop and there is not any other function call except for the check-points themselves. The aim of this program is to measure the time taken by each *CHECKPOINT()* to perform and save the single measurement. The final store-points has not been taken into account in the performance analysis result, since the processor's BPU (branch predictor unit) would probably fail the prediction of the last loop cycle (which has to exit the loop), leading to some small delay.

```
Max execution time: 69 ns
Min execution time: 46 ns
Avg execution time: 49.515151515151516 ns
```

These results are really significant and they prove that the overhead added by the framework is small and does not interfere with the performance of calling program.

13.9 Hints for programmers

13.9.1 Compiling a Polycube service

Polycube services can be compiled in two ways:

- together with the entire Polycube source code
- as stand-alone services that can be loaded into Polycube at runtime

Compiling together with the Polycube source code

1. Place the service into the src/services folder;
2. Update the src/services/CMakeLists.txt file adding a new *cmake* instruction:

```
add_service(service_name service_folder)
```

where the first argument is the service name used in the Polycube REST API, while second argument is the folder name of the service.

Note: the service folder name must follow the `pcn-service_name` naming convention (E.g. pcn-bridge, pcn-router).

3. Compile and re-install Polycube:

```
# having Polycube root folder as working directory
mkdir build && cd build
cmake ..
make -j $(getconf _NPROCESSORS_ONLN)
sudo make install
```

Compiling a stand-alone service

1. create a *build* folder in the root folder of the service and set it as working directory:

```
mkdir build && cd build
```

2. generate a Makefile:

```
cmake ..
```

3. compile the service to build a shared library object:

```
make -j $(getconf _NPROCESSORS_ONLN)
```

At this point the service has been compiled and the built shared library object can be found into the `build/src` folder named as `libpcn-service_name.so`

The compiled service, completely encapsulated in the .so file, is now ready to be loaded by the Polycube daemon.

13.9.2 Loading/unloading Polycube services at runtime

In order to load a new service at runtime, we should provide to Polycube daemon the shared library object (.so) we compiled before.

Loading a service

In order to load a new service into Polycube a shared library object must be provided. Please see *Compiling a stand-alone service* to generate this object.

To load a service into a running Polycube instance use:

```
polycubectl services add type=lib uri=/absolute/path/to/libpcn-service_name.so␣
↪name=service_name
```

- the `uri` parameter indicates the absolute path to the shared library object of the service;
- the `name` parameter indicates the name that will be use in the Polycube rest API for the service.

After having loaded the service, it can be instantiated as a normal service using `polycubectl`.

To verify the service has been loaded correctly and is ready to be instantiated:

```
# show available services list; the loaded service should be present
polycubectl services show

# create an instance of the service
polycubectl service_name add instance_name
```

(continues on next page)

(continued from previous page)

```
# dump the service
polycubectl instance_name show
```

Unloading a service

To unload a service from a running Polycube instance use:

```
polycubectl services del service_name
```

where `service_name` indicates the name used by the Polycube rest API for the service (E.g. bridge, router)

13.9.3 Install the provided git-hooks

An interesting feature that git provides is the possibility to run custom scripts (called *Git Hooks*) before or after events such as: commit, push, and receive. Git hook scripts are useful for identifying simple issues before submission to code review. We run our hooks on every commit to automatically point out issues in code such as missing semicolons, trailing whitespace, and debug statements.

To solve these issues and benefit from this feature, we use [pre-commit](https://pre-commit.com/), a framework for managing and maintaining multi-language pre-commit hooks.

The `.pre-commit-config.yaml` configuration file is already available under the root folder of this repo but before you can run hooks, you need to have the pre-commit package manager installed. You can install it using pip:

```
sudo apt-get install python-pip -y
pip install pre-commit
```

After that, run `pre-commit install` (under the project root folder) to install `pre-commit` into your git hooks. `pre-commit` will now run on every commit.

```
pre-commit install
```

13.9.4 How to write a test

The following is a brief guideline about how to write tests for services. Please remember such tests are invoked by a more generic script that tries to execute all tests for all services and provide global results.

1. tests are placed under *pcn-servicenametest* folder (and its first level of subfolders). E.g. *pcn-bridgetest* and *pcn-bridgetestvlan* are valid folders.
2. tests name begins with *test**
3. tests scripts must be executable (*chmod +x test.sh*)
4. never launch *polycubed*: polycubed is launched by the upper script, not in the script itself
5. exit on error: script should exit when a command fails (*set -e*)
6. tests must terminate in a fixed maximum time, no *read* or long *sleep* allowed
7. tests **must** exit with a **clean environment**: all *namespaces*, *links*, *interfaces*, *cubes* created inside the script must be destroyed when script returns. In order to do that use a *function cleanup* and set `trap cleanup EXIT` to be sure cleanup function gets always executed (also if an error is encountered, and the script fails).

8. consider that when `set -e` is enabled in your script, and you want to check if, for instance, a `ping` or `curl` command succeeds, this check is implicitly done by the returning value of the command itself. Hence, `ping 10.0.0.1 -c 2 -w 4` makes your script succeed if ping works, and make your script fail if it does not.
9. if the test *succeeded* it returns `0`, otherwise returns *non-zero-value* (this is the standard behavior). In order to check a single test result, use *echo $?* after script execution to read return value.

Please refer to existing examples (E.g. services/pcn-helloworld/test/test1.sh)

13.9.5 Optimizing the compilation time of dataplane eBPF programs

The compilation time of dataplane eBPF programs is an important parameter in services that are composed by many eBPF programs and/or services that use the reloading capability, as this affect the time needed to apply your changes in the data path.

Unfortunately, including large headers in the datapath code has a noticeable impact on the compilation time. Hence, in some cases it is better to copy-paste some elements (struct, macro, function, etc) and definitions to avoid including the whole file, or move to more specific headers instead of including everything from the root headers folder.

The *pcn-firewall* service is an example of service that has been optimized in this way, decreasing the dynamic loading time from tens of seconds to a few seconds.

13.9.6 Additional hints

1. **Creating multiple data plane programs**. If possible, it would be nice to create a single dataplane program, and enabling/disabling some portions using conditional compilation macros.
2. **Coding Style**: The `scripts/check-style.py` uses `clang-format` to check the code style. This tool has to be executed from the root folder. A list of files or folders to check is received as argument; the entire source code is checked when no parameters are passed. The `--fix` option will automatically fix the code style instead of simply checking
3. **Trailing white spaces**: Trailing white spaces could generate some git noise. Any decent text editor has an option to remove them automatically, it is a good idea to enable it. Please notice that running `clang-format` will remove them automatically.

Note: If you are using our `pre-commit git hooks`, you do not need to remove the trailing whitespaces manually, they will be removed automatically at every commit. If you want to remove them manually you can use execute the following commands in the Polycube root folder, please note that this will remove trailing whitespaces of all files.

```
find . -type f -name '*.h' -exec sed -i 's/[ \t]*$//' {} \+
find . -type f -name '*.cc' -exec sed -i 's/[ \t]*$//' {} \+
find . -type f -name '*.cpp' -exec sed -i 's/[ \t]*$//' {} \+
```

4. **Debugging using bpftool**: In some cases bpftool could be useful to inspect running eBPF programs and show, dump, modify maps values.

```
#install
sudo apt install binutils-dev libreadline-dev
git clone https://github.com/torvalds/linux
cd linux/tools
make bpf
cd bpf
sudo make install
```

(continues on next page)

(continued from previous page)

```
#usage
sudo bpftool
```

13.9.7 Continuous Integration

A continuous integration system based on *jenkins* is set-up for this project.

build pipeline is in charge to re-build the whole project from scratch at each commit. *testing* pipeline is in charge to run all repo system tests at each commit.

pipelines results are shown in *README* top buttons *build* and *testing*.

In order to get more info about tests results, click on such buttons to get redirected to jenkins environment, then click the build number to get detailed info about a specific commit. *testing/build* results can be read opening the tab *Console Output*.

What to do when *build failing* ?

Click in build failing button, go to jenkins control panel, open the build number is failing, open console output in order to understand where build is failing.

What to do when *testing failing* ?

Click in testing failing button, go to jenkins control panel, open the build number is failing, open console output in order to understand where tests are failing. At the end of the output there is a summary with tests passing/failing.

13.9.8 Valgrind

Valgrind is an open source tool for analyzing memory management and threading bugs. It can easily discover memory leaks, and spot possible segfault errors.

Requirements for polycubed: (1) valgrind 3.15+ (2) disable `setrlimit` in `polycubed.cpp`.

1. Install valgrind 3.15

Valgrind 3.14+ supports `bpf()` system call. Previous versions won't work.

- Download Valgrind 3.15+ source from here: http://www.valgrind.org/downloads/current.html
- Build Valgrind from source: http://valgrind.org/docs/manual/dist.install.html

```
./configure
make
sudo make install
```

2. Disable `setrlimit`

Only for debug purposes and in order to be able to run valgrind we have to disable `setrlimit` in `polycubed.cpp`.

We suggest to comment out following lines in polycubed.cpp

```
// Each instance of a service requires a high number of file descriptors
// (for example helloworld required 7), hence the default limit is too low
// for creating many instances of the services.
struct rlimit r = {32 * 1024, 64 * 1024};
if (setrlimit(RLIMIT_NOFILE, &r)) {
    logger->critical("Failed to set max number of possible filedescriptor");
    return -1;
}
```

13.10 Hateoas

Polycube supports the Hateoas standard. Thanks to this feature the daemon (*polycubed*) is able to provide the list of valid endpoints to the client that can be used to make requests to the server. The endpoints provided are those of the level just after the one requested. From the *polycubed* root you can explore the service through the Hateoas standard.

13.10.1 What can Hateoas be used for?

The Hateoas standard can be used to explore endpoints that can be used in Polycube. In this way a client will be able to access Polycube resources and thus cubes independently. This feature can also be useful in case you want to implement an alternative client to *polycubectl*.

13.10.2 How to use Hateoas

This feature can be used by observing the json of *polycubed* responses. You can use an http client (e.g. Postman) to execute requests to the Polycube daemon.

13.10.3 Example

In this example if the client request is

```
GET -> localhost:9000/polycube/v1/simplebridge/sb1/
```

then the server response will include all endpoints a of the level just after the simplebridge name (sb1).

```
{
  "name": "sb1",
  "uuid": "d138da68-f6f4-4ee9-8238-34e9ab1df156",
  "service-name": "simplebridge",
  "type": "TC",
  "loglevel": "INFO",
  "ports": [
      {
          "name": "tovethe1",
```

(continues on next page)

(continued from previous page)


```
            "uuid": "b2d6ebcb-163c-4ee8-a943-ba46b5fa4bda",
            "status": "UP",
            "peer": "veth1"
        },
        {
            "name": "tovethe2",
            "uuid": "e760a44d-5e9f-47e9-85d9-38e144400e83",
            "status": "UP",
            "peer": "veth2"
        }
    ],
    "shadow": false,
    "span": false,
    "fdb": {
        "aging-time": 300,
        "entry": [
            {
                "address": "da:55:44:5a:b1:b1",
                "port": "tovethe1",
                "age": 29
            },
            {
                "address": "11:ff:fe:80:00:00",
                "port": "tovethe2",
                "age": 26
            },
            {
                "address": "ee:c9:0e:19:c7:2a",
                "port": "tovethe2",
                "age": 1
            }
        ]
    },
    "_links": [
        {
            "rel": "self",
            "href": "//127.0.0.1:9000/polycube/v1/simplebridge/sb1/"
        },
        {
            "rel": "uuid",
            "href": "//127.0.0.1:9000/polycube/v1/simplebridge/sb1/uuid/"
        },
        {
            "rel": "type",
            "href": "//127.0.0.1:9000/polycube/v1/simplebridge/sb1/type/"
        },
        {
            "rel": "service-name",
            "href": "//127.0.0.1:9000/polycube/v1/simplebridge/sb1/service-name/"
        },
        {
            "rel": "loglevel",
```


(continues on next page)

(continued from previous page)


```
            "href": "//127.0.0.1:9000/polycube/v1/simplebridge/sb1/loglevel/"
        },
        {
            "rel": "ports",
            "href": "//127.0.0.1:9000/polycube/v1/simplebridge/sb1/ports/"
        },
        {
            "rel": "shadow",
            "href": "//127.0.0.1:9000/polycube/v1/simplebridge/sb1/shadow/"
        },
        {
            "rel": "span",
            "href": "//127.0.0.1:9000/polycube/v1/simplebridge/sb1/span/"
        },
        {
            "rel": "fdb",
            "href": "//127.0.0.1:9000/polycube/v1/simplebridge/sb1/fdb/"
        }
    ]
```


}

As we can see from this answer json, the "_links" section contains all the endpoints that the client can use to contact *polycubed* starting from the service name (sb1).

In this way a client can explore the service level by level.

13.10.4 How to know endpoints without hateoas?

Well, there is an alternative (harder) way to know the endpoints of a cube that can be used in a Polycube. The swaggerfile must be produced from the yang datamodel of a service (option **-s** in swagger-codegen).

To do this you have to provide the datamodel of a service as input to the swagger codegen (**-i** option). The swaggerfile is a large and not very understandable json. How can we study it in order to know APIs? We have to use the swagger-editor that can accept the swaggerfile generated and provides a more user friendly way to observe api's of a service. Swagger editor allows you to view endpoints and verbs that can be used to interact with a Polycube.

Note: using this method you will only know the endpoints of a cube and not all the endpoints offered by *polycubed*.

13.11 Polycube CI/CD

This guide is the starting point for developers that want to deal with CI/CD processes based on GitHub Actions.

13.11.1 Introduction

GitHub Actions help you automate tasks within your software development life cycle by means of workflows.

The workflow is an automated procedure that you add to your repository inside the .github/workflows/ directory. Are made up of one or more jobs and can be scheduled or triggered by an event.

The workflow can be used to build, test, package, release, and deploy a project on GitHub.

13.11.2 Implementation with polycube

You can find the workflow for CI/CD under .github/workflows/ci.yml.

This workflow can start with three different events for three different tasks:

1. **Pull Request (PR) opened/synchronized/reopened**: used to build and test any open PRs. The workflow assumes a "dev" state for its entire duration.
2. **Commit push on master (PR accepted)**: used to build and push an updated version of polycube on the public docker registry after a PR merge. In this situation the status of the workflow assumes the value of "master".
3. **Tag push on master (tag with format "v*")**: used to build and push a polycube release on the public docker registry with an official tag. The status of the workflow becomes "release" and a changelog is also published on github starting from the last release.

13.11.3 Workflow in "dev" state

This workflow is used to validate any open PR. For this reason, all tests are performed after the build. If there is only one failed test, the entire workflow will end in error, reporting it via annotation. Furthermore, it is possible to find the logs of the tests through the artifact section within the page of the performed workflow. In this state only the full polycube image will be built and published on **Docker Hub** under the `polycubenets/polycube-pr` repository.

13.11.4 Workflow in "master" state

This workflow starts once a PR is validated and accepted and it updates the _latest_ version of polycube by publishing the new image on `polycubenets` repository. Here the individual names change to polycube, polycube-iptables and polycube-k8s.

13.11.5 Workflow in "release" state

This workflow is used to release an official polycube tag. The new image will always be built and published on the `polycubenets` repository with the official names (polycube, polycube-iptables and polycube-k8s). It also provides for the creation of a release on GitHub by generating a changelog that summarizes all the accepted PRs starting from the previous release.

13.12 How to create a new service / update an existing one

The process to create or update service could be summarized in these steps:

1. *Write or update a datamodel for the service*
2. *Use the datamodel for generating a service stub or updating an existing service stub*
3. *Implement or update the eBPF datapath*
4. *Implement or update the control plane*

Please note that steps (1) and (2) are needed only when there is a change to the the YANG data model. In case we have to slightly modify an existing service (e.g., fixing a bug in the code), only steps (3) and (4) are required.