Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bandwidth Limit Enforcement for SCIONLab

Bachelor Thesis

Manuel Meinen

Advisors: Prof. Dr. Adrian Perrig, Mr. Matthias Frei
Department of Computer Science, ETH Zürich

Abstract

Today's society is heavily dependant on the internet and therefore also on it's underlying architecture. However, protocols like IP[1] and BGP[2] were not designed to withstand the threads that the internet is facing today. To address this issue, a research group around Prof. Adrian Perrig invented a novel internet architecture named SCION, which stands for *Scalability, Control and Isolation on Next-Generation Networks*.
For ISPs[3], research institutions or in general administrators of ASes[4] to test out SCION, the Network Security Group of ETH Zurich manages a distributed testbed called SCIONLab. Customers of SCIONLab can customize and download the configuration for setting up a SCION-AS. This SCION-AS connects then to one of ETH's Attachment Points and can therefore communicate via the SCIONLab infrastructure.
At the moment there are no bandwidth limitations in place other than the physical ones, meaning that neither the SCIONLab administrators nor the customers can set any upper limits for the bandwidth they have available between the User-AS and the SCIONLab infrastructure. However, such an upper limit is desirable for both customers and ETH for different reasons. Customers might want to test out the behaviour of a certain SCION based application when having limited bandwidth available. And ETH, which pays for the bandwidth SCIONLab is using, has an interest in enforcing an upper bandwidth limit to gain control over the expenses they have.
Since SCIONLab is an overlay network, meaning that whatever looks like a physical link from the perspective of a SCIONLab-AS is in fact a connection over the traditional IP based internet, bandwidth limitations per link can be enforced on an IP-level using existing tools.
The goal of this bachelor thesis project is to design and implement an automated mechanism to enforce a per IP-connection bandwidth limit between the User-ASes and the Attachment Points of SCIONLab. This is realized by using a tool called TC[5], which is part of the iproute2 utility collection.

[1] Internet Protocol
[2] Border Gateway Protocol
[3] Internet Service Providers
[4] Autonomous Systems
[5] Traffic Control

Contents

Introduction

Chapter 1

SCION/SCIONLab

1.1 SCION

SCION is a new internet architecture invented by Prof. Adrian Perrig and his fellow researchers. It's goal is to deliver scalability, control and isolation on next-generation networks. SCION differentiates between a control plane and a data plane, which are completely separated from each other to increase robustness. Failures in the control plane therefore don't immediately affect the availability of the data plane and vice versa.

1.1.1 Control Plane

Since SCION is an internet architecture it is designed to replace both IP and BGP and therefore fundamentally redesign the internet on OSI[1]-Layer 3 (Network Layer) and 4 (Transport Layer). Like the traditional internet, SCION is organized in Autonomous Systems. But unlike the traditional internet, SCION organizes multiple ASes in so called Isolation Domains. Per ISD[2], ASes have the same Trust Root Configuration.

Path Construction

Path construction in SCION happens in two phases, path exploration and path registration. In the path exploration phase the core-ASes send out PCBs[3]. The ASes can then decide to whom they want to forward the PCB. Each AS can then decide over which paths they want to be reached and register these paths in the path servers of the core-ASes. These processes happen both within an ISD as well as across ISDs. The final path is then constructed out of three partial paths, an up-path, a down-path and a core

[1] Open Systems Interconnection
[2] Isolation Domain
[3] Path-Segment Construction Beacons

path. How this final path is constructed is up to the AS itself, which gives it control over the path a certain package takes to reach it's destination. For further details I recommend reading chapter 7 of *SCION: A Secure Internet Architecture* [2]

1.1.2 Data Plane

The data plane consists mainly of path combination and data forwarding. Each AS possesses multiple up-paths. The down-paths are registered in the path servers of the core-ASes of the destination's ISD and can be requested by the AS. The core paths are stored in the path servers of the core-ASes. Therefore to get the possible paths to reach a destination AS, the source AS chooses an up-path to reach one of its core-ASes and requests the paths to one of the destination's core-ASes. Then it contacts the destination's core-AS to request the down-paths to the destination AS. Out of these three path segments the source AS can then construct the final path. By doing so it can also take short cuts over peering-links or if the destination AS lies on the up-path then it doesn't even need to contact any core-AS.
More details on how paths can be combined can be found in chapter 8 of the SCION-book [2]

1.1.3 Joining a SCION Network

For an AS to join a SCION network is quite easy. All the AS has to do is setting up at least one path server, one beacon server, one certificate server and one SCION border router. Then it buys connectivity from an AS that is already in the ISD that the AS wants to join as well. By joining the ISD the AS accepts the TRC configured for this ISD.

1.1.4 Topology

1.2 SCIONLab

SCIONLab is the distributed testbed for SCION. Most of the network is managed by the Network Security Group of ETH Zurich. The ASes that are not managed by ETH Zurich are called user-ASes. Each of them is typically attached to one of the few Attachment Points that are part of the network infrastructure managed by ETH. The APs are the only ASes that allow direct connections to user-ASes.
The SCIONLab network consists of multiple ISDs, most of which are grouping together ASes based on their geographic location or membership of a political union. However, one ISD has the purpose of building a backbone for the entire SCIONLab network. It is heavily interconnected and is hosted on Amazon Web Services.

1.2.1 SCIONLab as an Overlay Network

Each SCIONLab-AS is based on a Ubuntu-16.04 machine. Either it is a Virtual Machine or a dedicated SCION system. Each SCIONLab-AS has to at least have the following services available:

1. A **Beacon Server** that sends and receives the PCBs
2. A **Path Server** that stores the path segments and disseminates them to the customers.
3. A **Certificate Server** that holds the certificates which are used to validate the paths.
4. A **Border Router** that routes (on a SCION-level) the traffic leaving the AS.

As shown in figure 1.1 the Border Router is also responsible for wrapping the SCION-traffic into IP-traffic. This is simply done by setting up an IP-connection to the corresponding Attachment Point and then send the SCION packets over that connection. It is important to note that the Border Router doesn't do any routing on an IP-level.

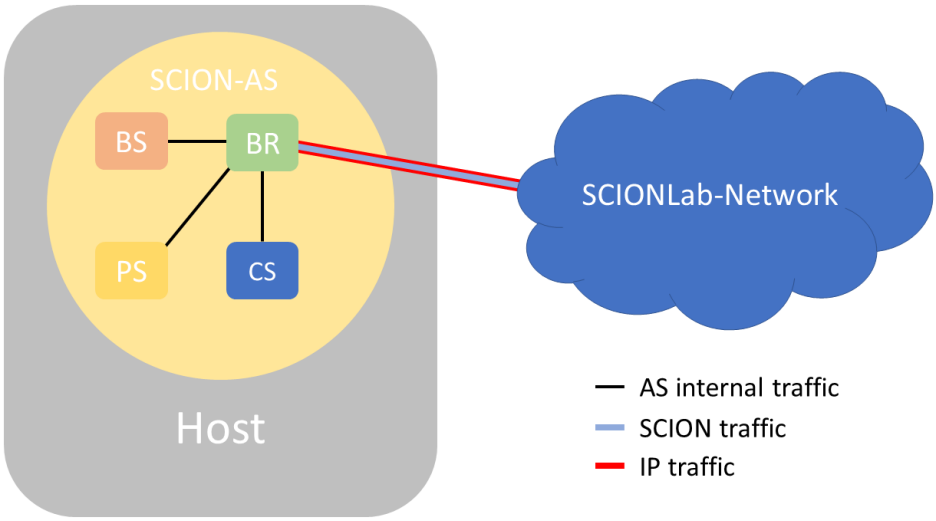

Figure 1.1: Schematic representation of a minimal SCIONLab-AS set-up

1.2.2 SCIONLab AS

There are two main ways to set up a SCIONLab-AS. The most common one is to set it up in a VM. The other option is to set it up on a dedicated SCION

system. Either way, you can do that by creating an account on scionlab.org and use the web interface to configure your AS to your needs. This web portal is called SCIONLab Coordination Service. After you configured your AS, you can download the files that are used to either set up the VM or the dedicated SCION system. The Attachment Point needs some time to receive the new configuration files to set up the connection to the user-AS. When the AP is ready, you will receive an e-mail that informs you about it. As visible in figure 1.2 the current version of the SCIONLab Coordination Service doesn't allow you to configure any bandwidth limits.

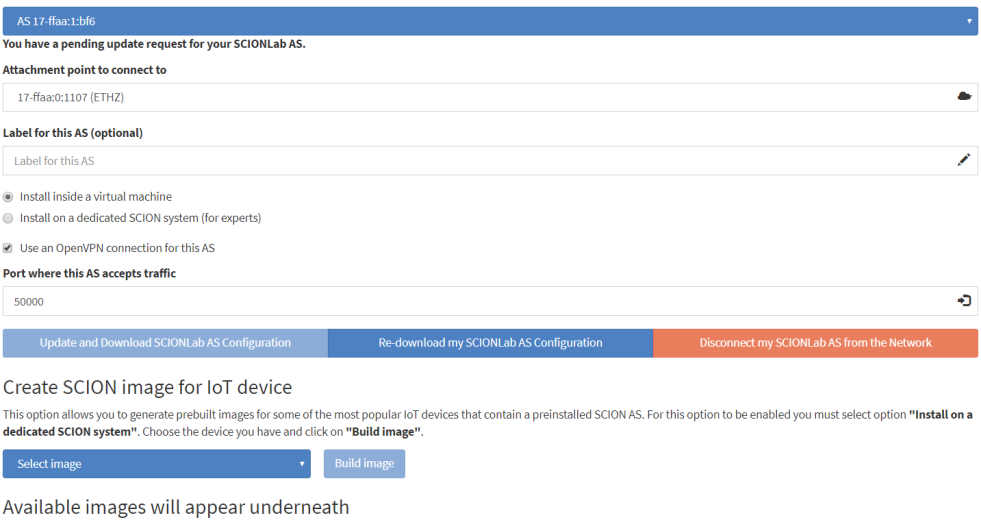

Figure 1.2: Web Interface of the SCIONLab Coordination Service

1.2.3 Topology

Chapter 2

Traffic Control

Traffic control in Linux is realized using a tool named TC. It consists of four basic techniques.

1. **Shaping**: This is the technique we will be using in order to enforce an upper bandwidth limit. But in general, it is the process of manipulating the bandwidth. It can also be used to smooth out bursts in order to improve the quality of service. Shaping is done on egress traffic.

2. **Scheduling**: This is the process of staging packets according to a schedule. Like this reordering of packets can be achieved. Scheduling as well happens with egress traffic.

3. **Policing**: This is the equivalent to shaping but for ingress traffic. It is worth noticing that traffic policing is more limited than traffic shaping, since there is no ingress queue.

4. **Dropping**: This is a quite primitive approach of just dropping traffic that exceeds a given bandwidth. This is applicable for both ingress and egress traffic.

For our needs, traffic shaping fits best. But since we need to limit ingress traffic as well, and shaping only operates on egress traffic, we need a workaround. More about that in section 2.1.3. Traffic control using TC is implemented using three basic building blocks: QDISCs[1], classes and filters. They will be discussed in section 2.2.

[1] Queuing Disciplines

2.1 Theory

2.1.1 Traffic Shaping

As previously stated, traffic shaping happens with egress traffic. However, there is a special QDISC that is handling ingress traffic. Since this QDISC is the only one applicable on ingress traffic, it is called *ingress*. The term Queuing Discipline, in that case, is a bit misleading, because there is no such thing as an ingress queue. All the other QDISCs handle egress traffic and are quite diverse. Each QDISC has a shaping algorithm that shapes the traffic passing the corresponding QDISC. QDISCs can be either classful or classless (the ingress QDISC is the latter). Classful QDISCs have a shaping algorithm that can handle traffic, which is classified in different classes, differently. Both the classes and the classifiers, which are filters that determine which kind of traffic belongs to which class, are attached to the QDISC. Classless QDISCs can obviously not handle classes. However they can handle filters, which can either do policing on the traffic or i.e. redirect it to a different interface.

2.1.2 Egress Traffic

To enforce a bandwidth limit per IP connection it makes sense to use a classful QDISC, because this provides us with the flexibility to configure the classes exactly according to our needs. The best choice for a classful egress QDISC that allows us to enforce bandwidth limits, seems to be an HTB[2]-QDISC, which is based on the TBF[3] shaping algorithm. The TBF algorithm roughly works as follows:
There is a bucket that holds tokens. Every token corresponds to approximately one byte. Whenever a packet arrives, the TBF algorithm tries to consume as many tokens out of the token bucket as there are bytes in the packet. If there are enough tokens, then the packet can be sent at full speed. If there are not enough tokens, then the packet gets enqueued and has to wait until there are again enough tokens in the bucket. The bucket is constantly being filled with tokens at the rate we configured. Like this the bandwidth can be limited in average and at the same time allow short bursts at maximum speed, where the size of these bursts depends on the size of the bucket. It is worth noticing, that the limitation looses in accuracy if the limit is above 1mbit/s. But the accuracy loss shouldn't be too severe for our use-case. More about the HTB-QDISC can be found in section 9.5.5. of *Linux advanced routing & traffic control HOWTO*[1].

[2]Hierarchy Token Bucket
[3]Token Bucket Filter

2.1.3 Ingress Traffic

There is only one QDISC that can handle ingress traffic, namely the ingress QDISC, which is classless. Since it is classless we can't do traffic shaping per IP connection as easily as we do it with egress traffic. Therefore we are left with two options:

1. **Traffic policing using filters**:
   With this option we can only use the configuration options TC-filters provide. The filters would be directly attached to the ingress QDISC. Since these policing options are a bit limited and not as powerful as the shaping options, we are not going to use policing.

2. **Traffic shaping using virtual interfaces**:
   The option of using virtual interfaces gives us the same configuration possibilities as we have with egress traffic. At the ingress QDISC we simply attach one filter that redirects all the traffic it receives to a virtual interface, on which we then have a HTB-QDISC and therefore have the same options as we have with the HTB-QDISC that handles egress traffic. You can think about this the following way: The virtual interface sends the ingress traffic to the host and therefore the ingress traffic becomes egress traffic from the perspective of the virtual interface. Figure 2.1 should make that clearer. Since this option is more powerful, in the sense that we have more shaping options, this is our option of choice.

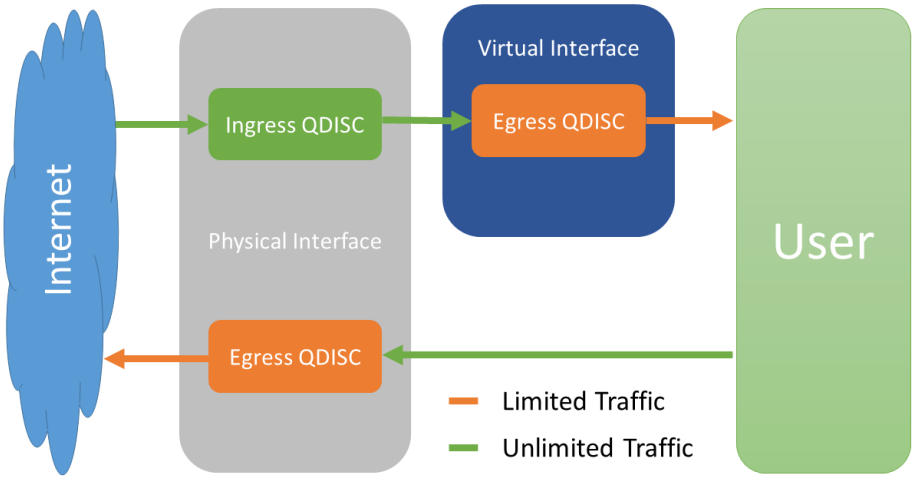

Figure 2.1: Interface set-up

2.2 TC

TC is a state of the art traffic control tool that is part of the iproute2 utility package and is by default installed on all Attachment Points. It can handle traffic based on any properties that can be found in the IP-header, as well as marks that were set using iptables or ipchains.

2.2.1 Queuing Disciplines

A QDISC or Queuing Discipline is the controlling unit that sits between the kernel and the interface. Whenever the kernel wants to send something to an interface, it enqueues it via the QDISC attached to the interface. The kernel then immediately tries to get as many packets as possible from the QDISC in order to send them to the network adapter driver. The QDISC can therefore decide in what manner it allows the kernel to get the packets back in order to send them. This also makes it clearer why there are very sophisticated egress QDISCs, while there is only one very primitive ingress QDISC.
As previously mentioned, QDISCs can be either classful or classless. If we want to do more than just simple policing or dropping then it is recommended to use classful QDISCs, simply because classful QDISCs can be configured more precisely, which makes them more powerful.

2.2.2 Classes

Classes are used to configure classful QDISCs. They are mostly used to handle a subset of the entire traffic. Which kind of traffic belongs to that subset is determined by classifier filters, which are attached to the parent QDISC. Each class has exactly one leaf QDISC, but can have multiple non-leaf QDISCs, to which there can again be classes attached. Like this classes and QDISCs can build up a hierarchy tree, which is then propagated from top to bottom, namely from the root QDISC to the leaf QDISC.

2.2.3 Filters

There are two main types of filters:

1. **Classifiers:** They map traffic based on some metric to a class of a classful QDISC, where the traffic is then further handled according to the leaf QDISC.
2. **Policers:** They are used to directly manipulate traffic. In our case we use them to redirect traffic from the ingress QDISC to a virtual interface. But they can also be used to directly limit the bandwidth, however without the configuration options a classful QDISC provides.

Chapter 3

Conception

In order to do a proper conception of the bandwidth limiter for SCIONLab, we have to formalize the requirements. We want to achieve the following:

- The SCIONLab administrators should be able to set a maximal bandwidth that is available to the user-ASes.
- The administrators of the user-ASes (the customers) should be able to set an upper limit for the bandwidth in the range of $[0, x]$ where x is the bandwidth limit set by the SCIONLab administrators.
- The above mentioned limits should be enforced automatically by only making configurations on the APs.

We split up these three requirements into two sections. The first two belong to the front end of the project, where as the third requirement is part of the back end.

3.1 Front end

The front end part of the project is all about the SCIONLab server. The implementation of that server can be found on github.com. There is already a configuration page, where the customers can set the desired configuration for their user-AS. On the same configuration page we simply add a field, where the customers can insert a bandwidth limit between 0 and the limit set by the SCIONLab administrators. Other bandwidth limits get already rejected by the form.
The SCIONLab server stores information about the entire topology of the SCIONLab network, including the links from the APs to the user-ASes. Per link there is a field that stores the maximal bandwidth for that link. This field exists but is currently not used. Therefore we simply set that attribute to the bandwidth limit that the customer chose.

The SCIONLab server regularly generates files according to its data model that are then sent to the APs, in order to configure them for newly added or updated user-ASes. These files are packed in a tarball and then delivered and unpacked on the APs. We can use this mechanism to deliver the information we need, in order to enforce the bandwidth limitations, to the APs. So we simply add a JSON[1]-file called *link_info.json* to that tarball, which holds the information we need to perform bandwidth limitations.

3.2 Back end

The back end part is all about configuring the APs in a way that the bandwidth limits that are specified in the *link_info.json* file are put into effect. We do that using the TC tool. However, there is no API to use TC in an automated way. Therefore we need to write a wrapper program, in order to configure TC via the command line. This wrapper program is written in Python and is available on github.com. Further details on how this program is implemented can be found in chapter 4.
Furthermore, we have to make some design decisions. Especially, we have to decide on how to build up the TC logic. Figure 3.1 shows in an abstract way how QDISCs, classes and filters are set up on both physical as well as virtual interfaces and how they are related.
Ingress, as well as egress traffic, firstly goes through the physical interface. Ingress traffic is then redirected by the *ingress* QDISC (with handle ffff) to a virtual interface. On the virtual interface, ingress traffic is handled the same way as egress traffic is on the physical interface. In both cases we use a HTB-QDISC. To make these QDISCs more distinguishable, we define the handle for a QDISC on a physical interface to be 1 and 2 if it is on a virtual interface. In both cases we define one class for each connection to a user-AS and set the corresponding bandwidth limit to its leaf-QDISC. Furthermore we define a default class, which limits all traffic going through this interface that doesn't match with any filter to the default bandwidth. Each class has as a class id the AS-id, which is unique to each AS. The default classes have a class id of 9999. Each HTB-QDISC has as many classifier filters as there are user-ASes plus one implicit one for the default class (represented as arrows in figure 3.1). The *ingress* QDISC has only one filter, which is a redirect filter that redirects ingress traffic of the physical interface to the virtual interface.

[1] JavaScript Object Notation

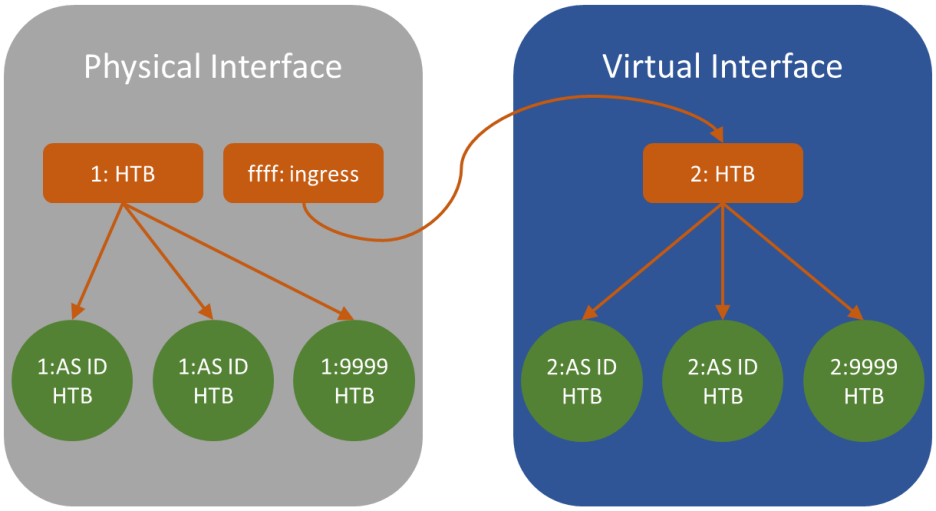

Figure 3.1: QDISC/Class hierarchy

Chapter 4

Implementation

4.1 Front end

4.2 Back end

The back end consists mainly of the python files listed below. Additionally there are some JSON-files that are used to either safe some settings or store some text that would take up too much space in the code and is therefore put into a separate file for the sake of readability. The python files have the following functionality:

- *scionlab_bw_limiter*:
  This file implements the entry point of the entire program. It parses command line arguments, let's you configure the default bandwidth and the path to the *link_info.json* file and invokes the *bandwidth_configurator.py* in order to enforce or reset bandwidth limitations or to show the current TC configuration. It accepts the following options:

  -h: Show help text that explains how to use the program. This text is stored in the *config_files/help.json* file.

  -l: Enforce the bandwidth limitations according to the *link_info.json* file.

  -r: Reset any previously set TC configurations.

  -s: Show the current TC configuration.

  -b: Set or update the default bandwidth. This takes as an argument a positive integer, which represents the default bandwidth in kilo bits per second.

  -p: Set or update the path to the *link_info.json* file. This option takes the path to that file as an argument.

- *code_base/bandwidth_configurator.py*:
  This file contains the implementation of the *limit()*, *reset()* and *show()* functions. The *limit()* function reads in the *link_info.json* file and creates link objects accordingly. Then it sets up the virtual interfaces, creates the TC logic and invokes the *make()* function from the *tc_logic.py* file. The *reset()* function simply deletes the root QDISC as well as the ingress QDISC for each used interface. And finally the *show()* function is responsible for printing out the current TC configuration.
- *code_base/cmd_executor.py*:
  The command executor implements some static helper functions, that simplify running a command on the command line. One function silently runs a command using the subprocess API, an other one runs a command and prints it to the console, one runs the command silently but returns the output it received by running the command and finally one function runs a command after it printed it and returns the output.
- *code_base/constants.py*:
  This file contains some static variables that are constant throughout the entire program.
- *code_base/interfaces.py*:
  This file is used to retrieve information about the interfaces configured on the host machine and bundle it together in interface objects.
- *code_base/links.py*:
  Analogously to the *interfaces.py* file the *links.py* file is used to bundle together information about links. This information is parsed form the *link_info.json* file. Furthermore, in this file the virtual interfaces are set up and mapped to their corresponding physical counterpart.
- *code_base/systeminfo.py*:
  The *systeminfo.py* file is used to retrieve some information about the host like whether a file exists or which interface is used by default.
- *code_base/tc_command_generator.py*:
  The TC command generator is used to generate the TC commands that are used in order to configure the system. The generation functions return these commands as strings. They are later executed using the command executor.
- *code_base/tc_logic.py*:
  This file is the centrepiece of the entire program. It defines the building blocks to build up the entire logic of our TC hierarchy according to figure 3.1. The UML[1]-model in figure 4.1 visualizes the relevant building blocks and their attributes and functions. The most interesting

[1] Unified Modeling Language

function is probably the *make()* function. It first uses the TC command generator to get the command to turn it's properties into TC configurations, executes the command using the command executor and then recursively calls *make()* on the building blocks that are lower down in the hierarchy tree. Like this the entire TC logic can be turned into TC configuration on the host.

- *code_base/virtual_interfaces_manager.py*:
  The virtual interface manager is used to set up and delete the virtual interface and to map them to their physical counterparts.

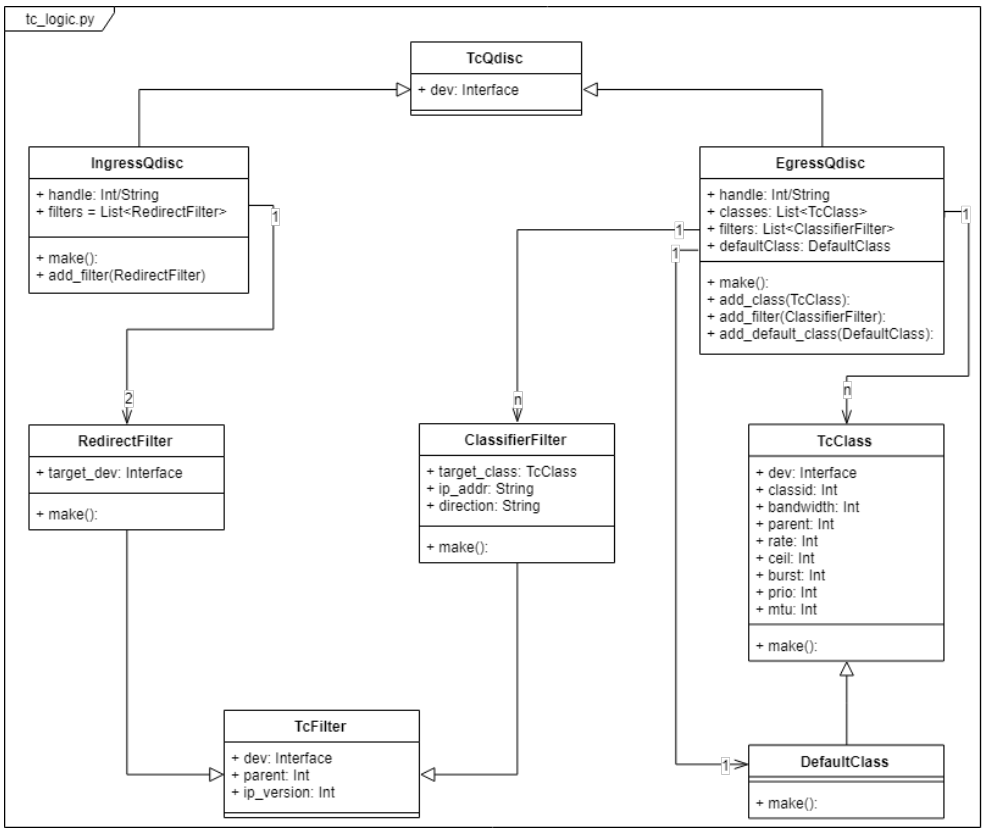

Figure 4.1: UML-Model of tc_logic.py

Chapter 5

Evaluation

5.1 iperf3

Chapter 6

Conclusion

Appendix A

Abbreviations

AP	Attachment Point
API	Application Programming Interface
AS	Autonomous System
AWS	Amazon Web Services
BGP	Border Gateway Protocol
BR	Border Router
BS	Beacon Server
CS	Certificate Server
ETH	Eidgenössische Technische Hochschule (Swiss Federal Institute of Technology)
HTB	Hierarchy Token Bucket
IP	Internet Protocol
ISD	Isolation Domain
ISP	Internet Service Provider
JSON	JavaScript Object Notation
OSI	Open Systems Interconnection
PCB	Path-Segment Construction Beacon
PS	Path Server
QDISC	Queuing Discipline
SCION	Scalability, Control and Isolation on Next-Generation Networks

SCIONLab	Testbed for SCION
TBF	Token Bucket Filter
TC	Traffic Control
TRC	Trust Root Configuration
UML	Unified Modeling Language
VM	Virtual Machine

Bibliography

[1] Bert Hubert et al. *Linux advanced routing & traffic control HOWTO*. 2002. URL: https://lartc.org/lartc.pdf (visited on 02/18/2019).

[2] Adrian Perrig et al. *SCION: a secure Internet architecture*. Springer, 2017.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.