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

This bachelor thesis project discusses the design, implementation and evaluation of an automated mechanism that enforces an upper bandwidth limit for the SCIONLab[1] network infrastructure.
The upper bandwidth limits are enforced on SCIONLab's Attachment Points and hold for both ingress as well as egress traffic between the AP[2] and the user-AS[3].
The backbone of this automated mechanism is implemented in Python, using the TC[4] utility of the iproute2 utility package.
The effectiveness of the implementation is evaluated using iPerf3, an open-source network performance measurement tool.

[1] Testbed for SCION
[2] Attachment Point
[3] Autonomous System
[4] Traffic Control

Contents

Chapter 1

Introduction

The Goal of this Bachelor Thesis Project

At the moment, on SCIONLab, there are no bandwidth limitations in place other than the physical ones, meaning that neither the SCIONLab administrators nor the users 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, the users as well as ETH, for different reasons. The users 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 utility called TC, which is part of the iproute2 utility collection.

Problems of the Current Internet

Today's society heavily depends 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 threats that the internet is facing today. To address this issue, a research group around Prof. Adrian Perrig invented a novel internet architecture named SCION[3], which stands for *Scalability, Control and Isolation on Next-Generation Networks*.

What is SCION?

SCION is a clean-slate network architecture. It provides route control, failure isolation and explicit trust information for end-to-end communication[1]. It is designed to replace both IP and BGP.

What is SCIONLab?

For ISPs[4], research institutions or in general administrators of ASes to test out SCION, the Network Security Group of ETH Zurich manages a distributed testbed called SCIONLab. Users 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.

[1] Internet Protocol
[2] Border Gateway Protocol
[3] Scalability, Control and Isolation on Next-Generation Networks
[4] Internet Service Providers

Chapter 2

SCION/SCIONLab

2.1 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.

2.1.1 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, the path exploration phase and the path registration phase. In the path exploration phase the core-ASes send out PCBs[3]. The ASes can then decide to whom they want to forward the PCBs. Each AS can then decide over which paths they want to be reachable 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

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

final path is then constructed out of three partial paths, an up-path, a down-path and a core 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 its destination. For further details I recommend to read chapter 7 of *SCION: A Secure Internet Architecture* [2]

2.1.2 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 any 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 ISD'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]

2.1.3 Joining a SCION Network

For an AS it is quite easy to join a SCION network. 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.

2.1.4 Topology

Figure 2.1 shows an example of a basic SCION topology. In this example we have two Isolation Domains. One ISD is managed by three, the other one by two core-ASes. What this figure doesn't show, is that ISDs can be overlapping.

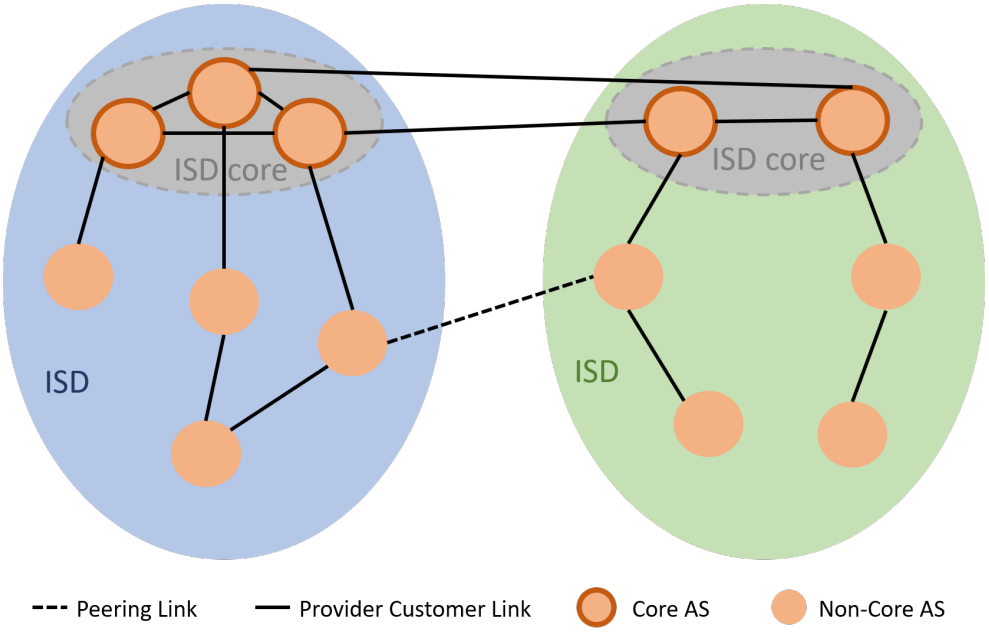

Figure 2.1: Basic SCION Topology

2.2 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.

2.2.1 SCIONLab as an Overlay Network

Each SCIONLab-AS is based on a VM (currently running Ubuntu-16.04), a container or a dedicated SCION system. Each SCIONLab-AS has to at least have the following services available:

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

As shown in figure 2.2 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.

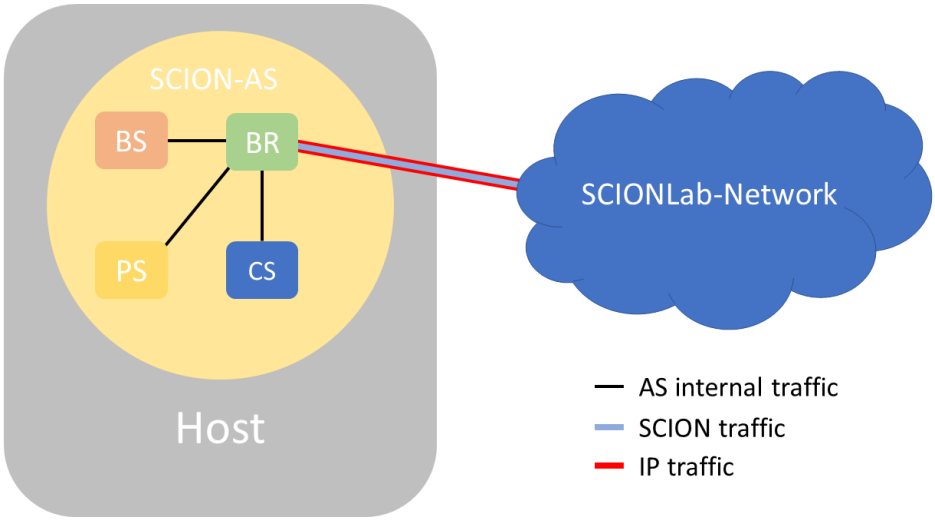

Figure 2.2: Schematic representation of a minimal SCIONLab-AS set-up

2.2.2 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 2.3 the current version of the SCIONLab Coordination Service doesn't allow you to configure any bandwidth limits.

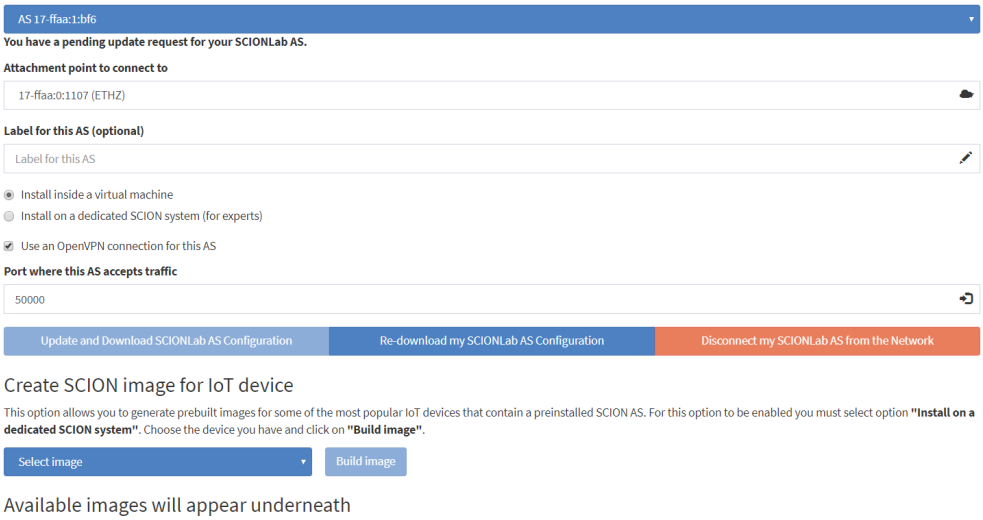

Figure 2.3: Web Interface of the SCIONLab Coordination Service

2.2.3 Topology

Figure 2.4 shows how the SCIONLab-Network is set up. The user-ASes are hosted by the users and connect to SCIONLab by setting up a connection to an AP. By doing so, the user-AS automatically joins the Attachment Points ISD. Infrastructure-ASes are ASes that are permanently connected to the SCIONLab-Network and take part in routing and forwarding packets according to the SCION protocol (APs are infrastructure-ASes as well). Even though it's not shown in figure 2.4 the ASes of SCIONLab are of course also grouped into ISDs. The only difference between Attachment Points and other infrastructure-ASes is that APs allow connections to user-ASes, while the normal infrastructure-ASes don't.

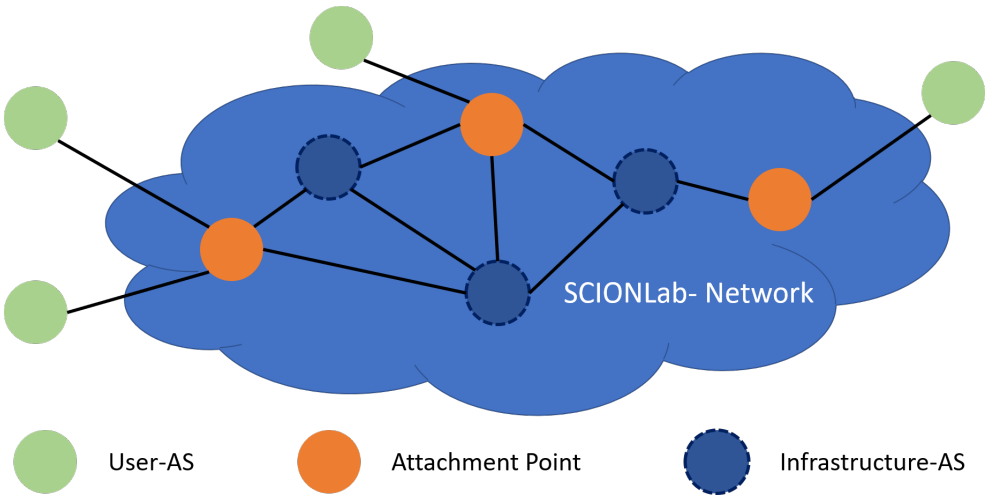

Figure 2.4: Basic SCIONLab Topology

Chapter 3

Traffic Control

Traffic control in Linux is realized by using a utility named TC. It consists of four basic techniques[3].

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 of 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 3.2.2. Traffic Control on Linux is implemented using three basic building blocks: QDISCs[1], classes and filters. They will be discussed in section 3.1.

[1]Queuing Disciplines

3.1 TC

TC is a state of the art traffic control utility 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.

3.1.1 Queuing Disciplines

A QDISC or Queuing Discipline is the controlling unit that sits between the kernel and the network 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. Therefore, it is clear that there are very sophisticated egress QDISCs, while there is only one very primitive ingress QDISC. As we will see in section 3.2, 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.

3.1.2 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.

3.1.3 Filters

There are two main types of filters[3]:

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 or filters attached to the class.
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 (IFB[2]). But they can also be used to directly limit the bandwidth, however without the configuration options a classful QDISC provides.

[2]Intermediate Functional Block

3.2 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 redirect it to a different interface.

3.2.1 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[3]-QDISC, which is based on the TBF[4] 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 1Mbps. 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*[3].

[3]Hierarchy Token Bucket
[4]Token Bucket Filter

3.2.2 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 3.1 should make that clearer. Since this option is more powerful, in the sense that we have more shaping options and because using the same egress QDISC on the virtual interface as we use on the physical interface gives us the advantage that the entire set-up is more symmetric and therefore easier to understand and debug, this is our option of choice.

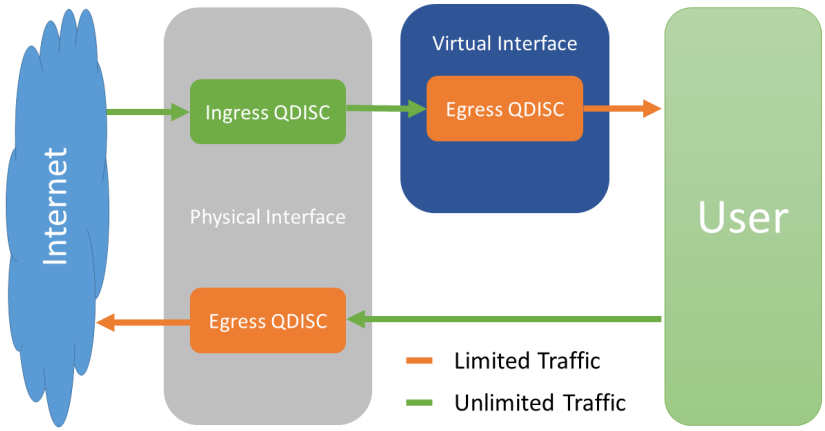

Figure 3.1: Interface set-up

Chapter 4

Conception

In order to conceptualize the bandwidth limiter for SCIONLab properly, we first need 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 users 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.

4.1 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[4]. There is already a configuration page, where the users can set the desired configuration for their user-ASes. On the same configuration page we simply add a field, where the users 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 information about 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 already exists but is currently not used. Therefore we simply set that attribute to the bandwidth limit that the user 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 to the APs, in order to enforce the bandwidth limitations. 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.

4.2 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 utility. However, there is no API to use TC in an automated way. Therefore we need to write a wrapper program that configures TC via the command line. This wrapper program is written in Python and is available on github.com[5]. Further details on how this program is implemented can be found in chapter 5.
Furthermore, we have to make some design decisions. Especially, we have to decide on how to build up the TC logic. Figure 4.1 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 an 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 the AS-id as a class 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 4.1). 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

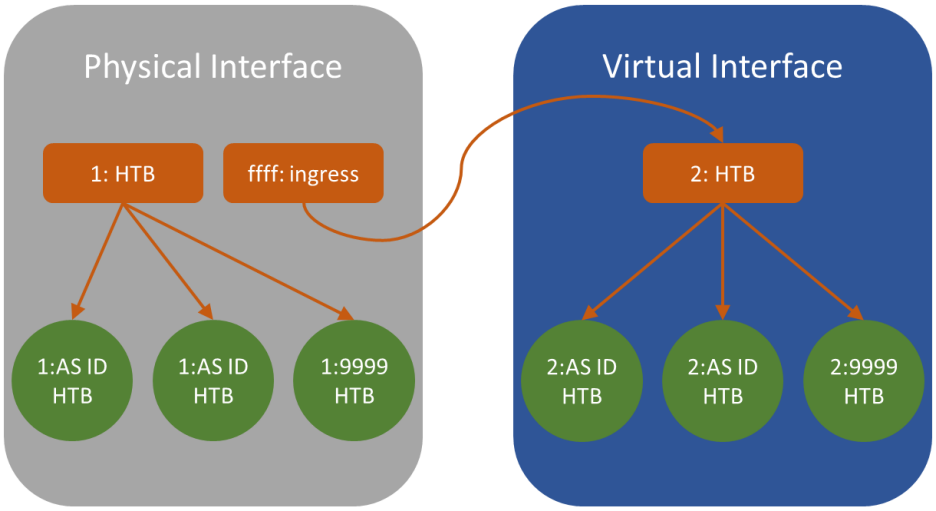

Figure 4.1: QDISC/Class hierarchy

Chapter 5

Implementation

5.1 Front End

5.1.1 The SCIONLab Website

In the configuration form for user-ASes of SCIONLab's web interface, I added an additional text input field (as visible in figure 5.1) that allows the user to add an upper limit for the bandwidth. If the user then clicks the **Create AS** button, the input from that field is read into the data model. Updating the configuration of an existing AS works similarly. The maximum allowed bandwidth limit is the default bandwidth, which is set by the SCIONLab administrators.

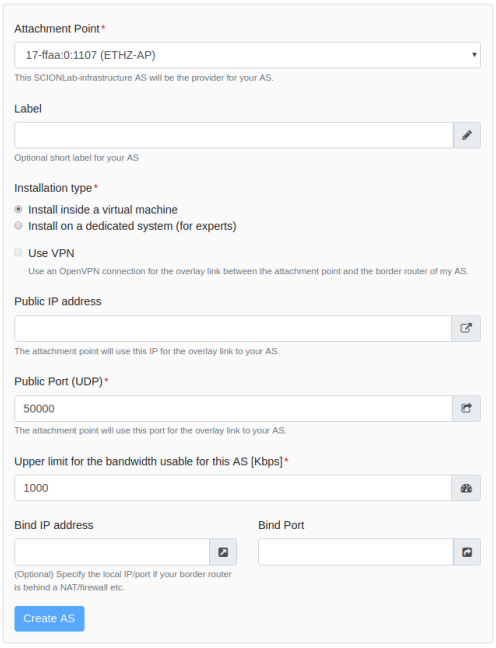

Figure 5.1: User-AS Configuration Form

5.1.2 The Data Model

The SCIONLab server is based on the Django framework. Therefore, SCIONLab's data model is based on Django's data model as shown in figure 5.2. The bandwidth limit is stored as an attribute in the *Link* class. As visible in figure 5.2, the *Host* class can access all of its interfaces and therefore also all of its links. The *Host* class represents the VM, container or physical machine hosting the SCION services. Since the host can access all information we need to generate the *link_info.json* file, we hand a host object as an argument into the function *generate_link_info(host)* of the file *bandwidth_config.py*. This function returns a dictionary which is used in the *config_tar.py* file to generate a JSON file and add the *link_info.json* file to the tarball that is later deployed to the Attachment Point.

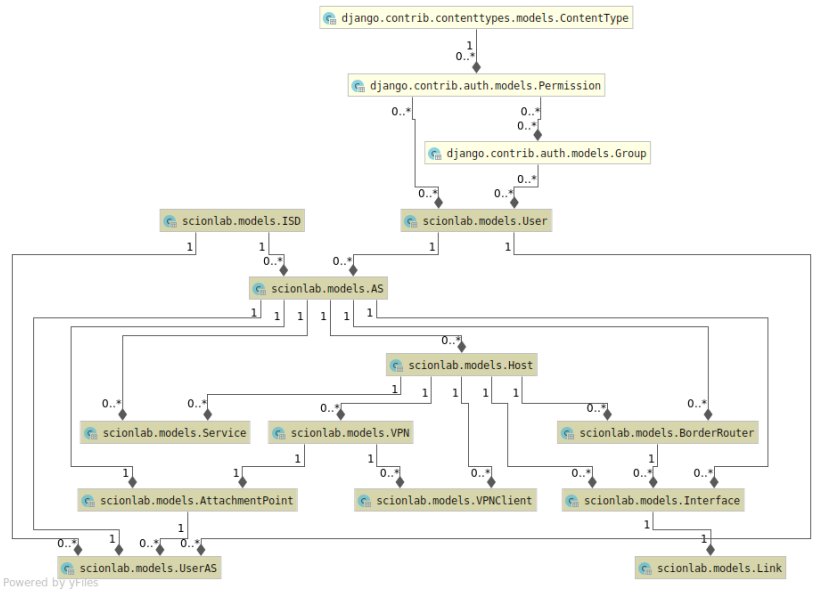

Figure 5.2: Django Model Dependency Diagram

5.1.3 Attachment Point Configuration

On the Attachment Points will be a Python file called *scionlab_config.py*. This file is responsible for configuring the APs and to restart the SCION services after the configurations changed. Therefore, whenever we restart the SCION services, we also have to invoke the *scionlab_bw_limiter* (with the *-l* option) in order to modify the TC configuration according to the current *link_info.json* file. It is important to note that TC configurations get automatically reset after each reboot and therefore need to be re-enforced every time the AP restarts.

5.2 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 functionalities:

- *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, a third one runs the command silently but returns the output it received by running the command

and finally one function runs a command after printing 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 into 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 from 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 4.1. The UML[1]-model in figure 5.3 visualizes the relevant building blocks and their attributes and functions. The most interesting function is probably the *make()* function. It first uses the TC command generator to get the command to turn its 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 interfaces and to map them to their physical counterparts.

[1] Unified Modeling Language

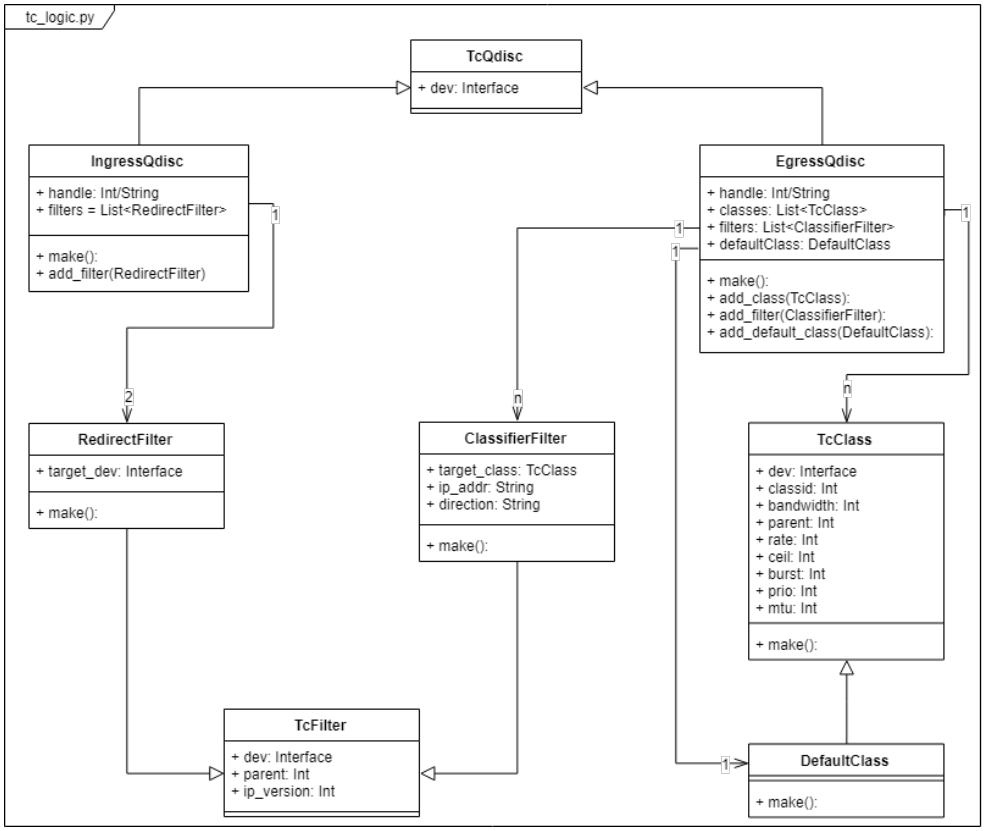

Figure 5.3: UML-Model of tc_logic.py (UML 2.0)

5.2.1 Configuration Example

This example shows how the *link_info.json* file gets turned into TC configuration using the *scionlab_bw_limiter*. Listing 5.1 shows the entries for two user-ASes. The first one is connected to the AP via an IPv6 connection, whereas the second user-AS uses an IPv4 connection. Both ASes have a bandwidth limit of 500 Kbps and the default bandwidth is 1000 Kbps. Listing 5.2 shows how TC gets configured accordingly. Note that the *scionlab_bw_limiter* also needs to set up the virtual interfaces. But since this is not really dependant on the *link_info.json* file, I omitted this part for the sake of readability. Furthermore, it is worth noticing that TC needs to be executed with root privileges, therefore in order to run the commands from listing 5.2 simply prepend the *sudo* command.

```
1  {
2      "1": {
3          "AS_ID": 1,
4          "IsUserAS": true,
5          "Bandwidth": 500,
6          "IP-Address": "fe80::20c:29ff:fe08:993c"
7      },
8      "2": {
9          "AS_ID": 2,
10         "IsUserAS": true,
11         "Bandwidth": 500,
12         "IP-Address": "192.168.17.129"
13     }
14 }
```

Listing 5.1: link_info.json

```
1  # Create root QDISC on the physical interface
2  tc qdisc add dev ens33 root handle 1: htb default 9999
3  # Create classes
4  tc class add dev ens33 parent 1:0 classid 1:9999 htb
       rate 1000kbit ceil 1000kbit burst 5k prio 9999 mtu
       1500
5  tc class add dev ens33 parent 1:0 classid 1:1 htb rate
       500kbit ceil 500kbit burst 5k prio 1 mtu 1500
6  tc class add dev ens33 parent 1:0 classid 1:2 htb rate
       500kbit ceil 500kbit burst 5k prio 2 mtu 1500
7  # Create classifier filters
8  tc filter add dev ens33 u32 match ip6 dst
       fe80::20c:29ff:fe08:993c/128 flowid 1:1
9  tc filter add dev ens33 u32 match ip dst
       192.168.17.129/32 flowid 1:2
10 # Create ingress QDISC on the physical interface
11 tc qdisc add dev ens33 handle ffff: ingress
12 # Create redirect filters
13 tc filter add dev ens33 parent ffff: protocol ip u32
       match ip src 0.0.0.0/0 action mirred egress redirect
       dev ifb0
14 tc filter add dev ens33 parent ffff: protocol ipv6 u32
       match ip6 src ::0/0 action mirred egress redirect
       dev ifb0
15 # Create root QDISC on the virtual interface
16 tc qdisc add dev ifb0 root handle 2: htb default 9999
17 # Create classes
18 tc class add dev ifb0 parent 2:0 classid 2:9999 htb
       rate 1000kbit ceil 1000kbit burst 5k prio 9999 mtu
       1500
19 tc class add dev ifb0 parent 2:0 classid 2:1 htb rate
       500kbit ceil 500kbit burst 5k prio 1 mtu 1500
20 tc class add dev ifb0 parent 2:0 classid 2:2 htb rate
       500kbit ceil 500kbit burst 5k prio 2 mtu 1500
21 # Create classifier filters
22 tc filter add dev ifb0 u32 match ip6 src
       fe80::20c:29ff:fe08:993c/128 flowid 2:1
23 tc filter add dev ifb0 u32 match ip src
       192.168.17.129/32 flowid 2:2
```

Listing 5.2: TC Configuration

Chapter 6

Evaluation

Evaluating the results of bandwidth limitations is more difficult than it might seem. Both, bandwidth limitation utilities like TC as well as testing tools like iPerf3 are often hard to configure correctly and are sometimes documented quite incomprehensibly. Therefore, it is often unclear whether errors are measurement errors or limitation errors. Furthermore, there is no such thing as standard IP- or even UDP[1]-traffic. Packets can have different sizes, can be reordered or get lost. All those factors can influence and sometimes falsify the measurement. And finally it is questionable whether the traffic that is generated by the testing tools is similar enough to normal traffic generated by an AS, such that we can conclude anything meaningful. What is known about the environment in which the *scionlab_bw_limiter* will be running is that we deal with UDP-traffic over IPv4 and at some point in the future might be dealing with IPv6 traffic as well. Therefore, we mainly focus on testing the bandwidth limitations using UDP-traffic over IPv4. However, note that the *scionlab_bw_limiter* also works with IPv6 traffic and that the test results look almost identical to the test results when using IPv4 traffic.
In this chapter we will first discuss the general test set-up, then a first test approach will be presented, which shows some unexpected anomalies. In section 6.3.3, we will examine the cause for these anomalies. From the knowledge gained from this examination, we deduce a realistic test configuration that delivers the results that we expect and that proves the effectiveness of the *scionlab_bw_limiter*. The final results are presented in figure 6.8.

[1]User Datagram Protocol

6.1 Test Set-up

As a test set-up it makes sense to have a test server, which doesn't have any bandwidth limitations and a test client, where the *scionlab_bw_limiter* will enforce the desired bandwidth limits. As a test client I decided to use my development VM, which I also use to develop and test the *scionlab_bw_limiter*. This VM runs Ubuntu 18.04, which is a newer version than the one running on the Attachment Points, but since TC works the same way on both versions of Ubuntu, this should not matter. The test server is a Ubuntu 18.04 server that runs in a VM as well and is hosted on the same physical machine as the test client. Therefore traffic between the test server and the test client is not real network traffic, but since the TC configurations are effective between the Linux kernel and the network driver, where both of which are virtualized, the TC configurations should have the exact same effect in this setting as in the real environment.

6.1.1 Set-Up Using iPerf3

I installed iPerf3 on both the test client as well as on the test server and let the test server run iPerf3 as a server. Note that the same software is used for both the server side as well as the client side. They just run in different modes. On the client I can now connect to the test server and run highly customisable tests with the server. Normally the client is the one sending data to the server, but iPerf3 can be run in a reverse mode (using the *-R* option), such that we can test both ingress as well as egress traffic without having to switch between client and server mode on our machines.

6.2 iPerf3

iPerf3 is a state of the art network performance measurement tool. It is a successor of iPerf and iPerf2. However it has been completely rewritten in order to make the code base cleaner and simpler and is therefore not backward compatible with iPerf2. iPerf3 has been developed by ESnet[2], which is a high-speed computer network provider for the United States Department of Energy. iPerf3 is open-source and can be found on github.com[6].
iPerf3 can test both TCP[3] as well as UDP traffic. In our case we only need to test UDP traffic. This can be done by passing the option *-u* as an argument to iPerf3. When running it we need to specify a target bandwidth (option *-b*). iPerf3 then tries to achieve this bandwidth by sending traffic out at this rate. To run iPerf3 in server mode we only have to pass *-s* as an argument. For running it in the client mode we have to analogously pass the *-c* option

[2]Energy Sciences Network
[3]Transmission Control Protocol

followed by the server's IP-address. Last but not least, we can optionally set the buffer length of the buffer from which is sent and to which is received by passing the option *-l* followed by a size in bytes as an argument. This will determine the size of the datagrams that are being sent. As we will see in section 6.3, this option is of quite a significance.

6.3 Test Results

The following test results occurred under the following configuration:
The server's IP-address is in our case 192.168.17.129, the IP-connection to the server is limited to 500Kbps and the default bandwidth is 1000Kbps. The MTU[4] parameter in TC is set to the MTU of the network interface that is used to connect to the specific IP-address. In our case this is 1500 bytes. The burst parameter, which is the size of the bucket in the TBF algorithm is set to 5K. The ceil rate is the same as the normal rate, which is 500Kbps to the test server and 1000Kbps to any other IP-device that uses the same interface.

On the test server, we start iPerf3 using no other options than the *-s* option, that makes iPerf3 run in server mode (see listing 6.1). On the client side, we configure a bit more. We set the client into client mode using the *-c* option followed by the server's IP-address. Furthermore, we set the *-u* option, in order to generate UDP traffic as test traffic. The target bandwidth is set to 2Mbps (option *-b*) and the only parameter that we are going to change for different tests is the datagram size (option *-l*). We start at a size of 1000 bytes and go up to 10000 bytes. And finally we set the *-R* option in case we want to run the test in reverse mode, meaning that we test with ingress traffic. Listings 6.2 and 6.3 show what these commands look like. Note that if we set the datagram size too small, we encounter an error. This happens because the packets arrive in a different order than they were sent. Therefore iPerf3 can't measure the bandwidth any more. This phenomenon is discussed on github.com[7] in the issues section of the iPerf repository.

[4]Maximum Transmission Unit

```
1 iperf3 -s
```

Listing 6.1: Test Server Command

```
1 iperf3 -c 192.168.17.129 -u -b 2Mbit -l 1000
```

Listing 6.2: Example Egress Test Command

```
1 iperf3 -c 192.168.17.129 -u -b 2Mbit -l 1000 -R
```

Listing 6.3: Example Ingress Test Command

6.3.1 Interpretation of the Test Results

Figure 6.1 shows the average bandwidth over ten samples for both ingress as well as egress traffic with respect to the datagram size. For the egress traffic the results look quite good. The average egress bandwidth lies slightly above the bandwidth limit. This is because we allow short bursts at a higher speed. Therefore, the limitations on egress traffic can be considered as effective and stable.
However, for ingress traffic it is a bit a different story. That the ingress bandwidth is lower than the egress bandwidth is expected, because there is a slight overhead caused by redirecting ingress traffic from a physical interface to a virtual interface. So having a situation like we have it with a datagram size of 1000 bytes would be desirable. However, we can't ignore that by increasing the datagram size, the ingress bandwidth drops quite drastically. For getting on the bottom of this issue, let us consider figure 6.2. This figure shows the number of samples at a bandwidth of zero depending on the size of the datagram. It is visible that the bigger the datagram, the more the ingress traffic suffers from samples with a bandwidth of zero. This phenomenon happens with egress traffic as well, but there it is much less significant. The increasing number of zero bandwidth samples shows that the decreased average bandwidth is not caused by a wrong limitation, since the samples that make it through are received at a bandwidth that is around the desired limit, but that for some samples either iPerf3 fails to do a measurement at all or something prevents the machine from receiving data for some time. To figure out whether these anomalies are the result of side effects the *scionlab_bw_limiter* causes or are measurement errors of iPerf3 or reflect normal behaviour of traffic under the current test configuration, we need to compare these results with results we get if we enforce a bandwidth limit using a different tool.

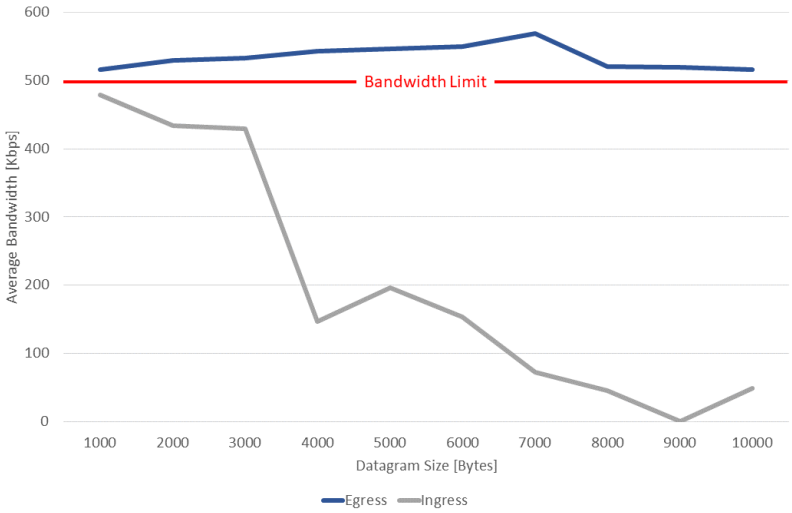

Figure 6.1: Evaluation of the Bandwidth

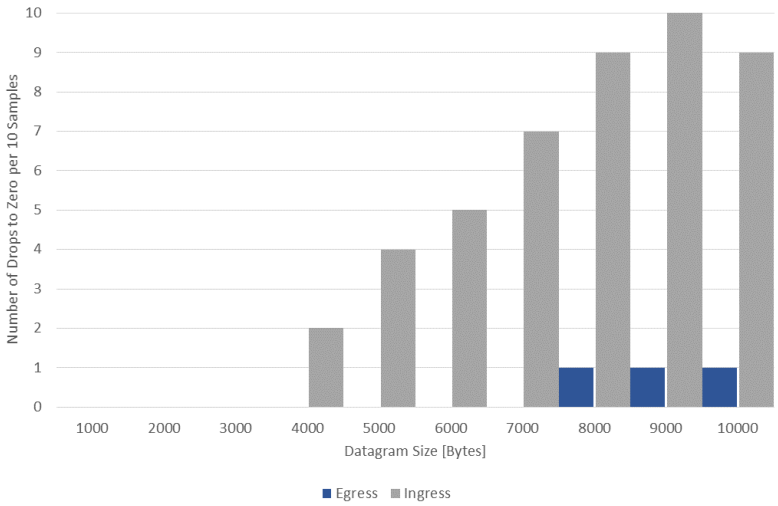

Figure 6.2: Evaluation of Bandwidth Drops

6.3.2 Comparison with Wondershaper

Wondershaper is an open-source program that has been developed by some of the major contributors of the *Linux advanced routing & traffic control HOWTO*[3]. The code of *wondershaper* can be found on github.com[8]. *Wondershaper* only allows the user to enforce a general bandwidth limit and not individual bandwidth limits per IP-address. On the other hand it allows us to set an ingress bandwidth that is different from the egress bandwidth. But since we don't require that, we just set the same bandwidth limits in both cases on the interface that is used to connect to the test server. Listing 6.4 shows the command that has been used in order to enforce a bandwidth limit of 500Kbps on interface *ens33*.

```
1 sudo ./wondershaper -a ens33 -u 500 -d 500
```

Listing 6.4: Wondershaper command

As we can se in Figure 6.3 and 6.4 the same phenomenon occurs if we enforce a bandwidth limit with *wondershaper* and run the exact same tests as before. It is worth noticing that even though the test results for ingress traffic look similarly bad, yet quite different from the results from the *scionlab_bw_limiter*, both *wondershaper* as well as the *scionlab_bw_limiter* use the IFB interfaces and the *ingress* QDISC and are therefore implemented almost equivalently. The egress bandwidth limits however are implemented quite differently even though the test results look almost the same. When it comes to egress traffic, *wondershaper* distinguishes between different types of service, in order to achieve a better quality of service. In our case this is not necessary, since we are not dealing with average "every day traffic", but only with SCION traffic that is wrapped into IP packets.

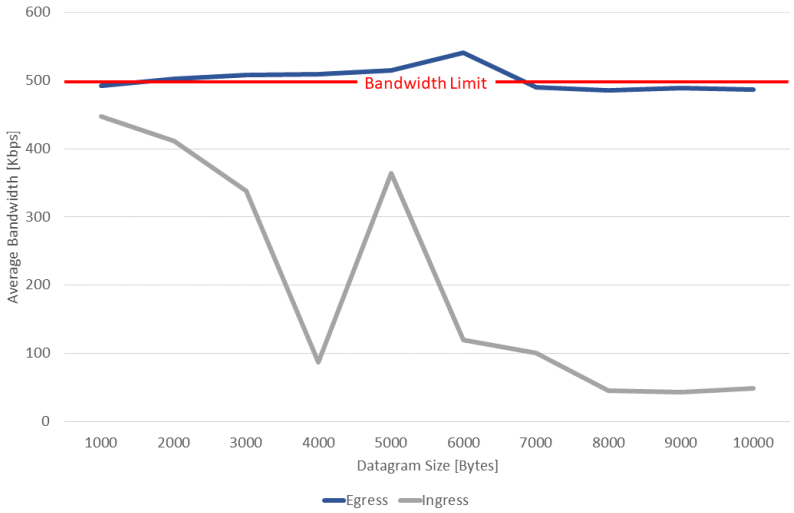

Figure 6.3: Evaluation of the Bandwidth (Wondershaper)

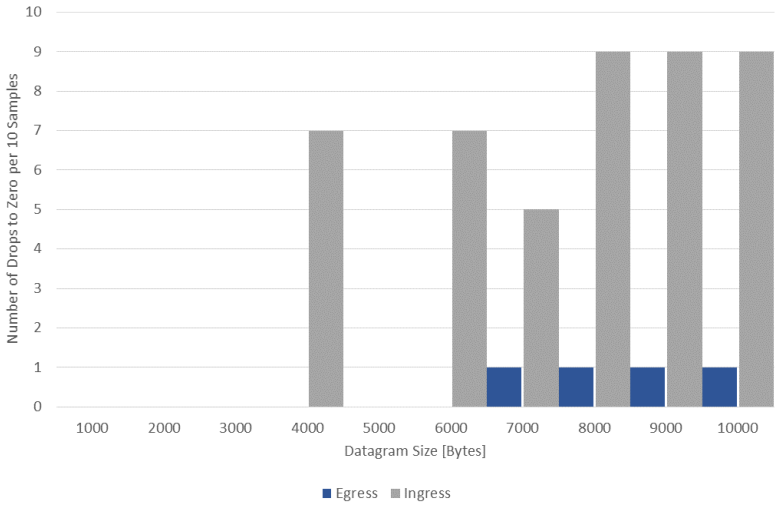

Figure 6.4: Evaluation of Bandwidth Drops (Wondershaper)

6.3.3 Examination of the Problem with Ingress Traffic

If the size of a datagram that is being sent by iPerf3 in order to determine the bandwidth is bigger than the MTU, then fragmentation happens. The Internet Protocol itself doesn't handle the loss of packets. This is the responsibility of protocols higher above in the OSI model. In our case, we have UDP as a OSI-layer 4 protocol. In UDP, if a fragment gets lost, then the entire packet is considered lost. Bandwidth limitations using TC on egress

traffic happen before the datagrams get fragmented and on ingress traffic, the reassembly of the fragments happens after the TC configurations had their effect (see figure 6.5 for further clarification). This leads to the hypothesis that if iPerf3 sends datagrams that need to be fragmented and tries to send them at a speed that is higher than the enforced bandwidth limit, we measure a significant drop in bandwidth, because some fragments of the datagram get dropped and therefore the entire datagram gets dropped.

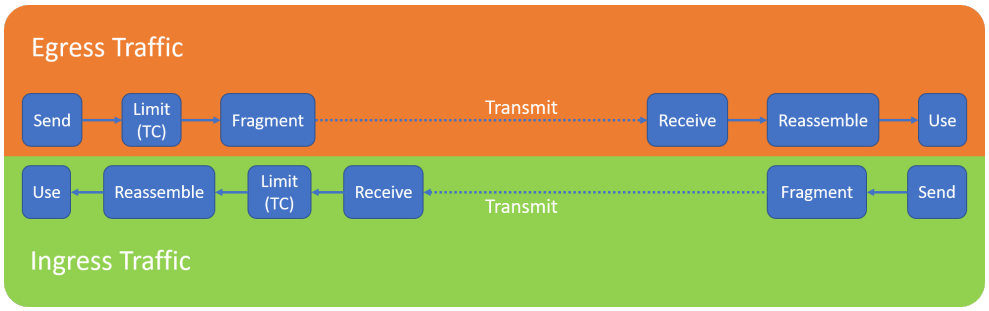

Figure 6.5: Schematic Representation of where Fragmentation occurs

To figure out whether the previously mentioned hypothesis holds, we again run two different tests.
This time we fix the datagram size and vary the target bandwidth from 100 Kbps to 1000 Kbps (option *-b*). The upper limit for the bandwidth is still 500 Kbps and has been enforced using the *scionlab_bw_limiter*.
For the first test, we set a datagram size of 10000 bytes, while having a MTU of 1500 bytes. Therefore, fragmentation is going to happen in this case. Figure 6.6 shows the results of this test. It is visible that as long as the target bandwidth is below the bandwidth limit of 500 Kbps, ingress traffic reaches the desired target bandwidth. However, as soon as we try to send traffic at a higher rate than the enforced limit, we see the significant drop in bandwidth that we observed beforehand. As figure 6.7 shows, a lot of samples experience a bandwidth drop to a bandwidth of zero as soon as they try to reach a bandwidth above the enforced limit. For egress traffic, iPerf3 fails to reach such a low target bandwidth, because it tries to send at least 10 datagrams per second, with each datagram having the specified size. Therefore, the effective bandwidth for egress traffic is:

$$\text{effective bandwidth} = max\left\{\frac{10 \cdot \text{datagram size}}{1}\left[\frac{bytes}{second}\right], \text{target bandwidth}\right\}$$

The fact that ingress traffic behaves as expected if we have a target bandwidth which is below the enforced limit, regardless of the datagram size, shows that the enforced limit is responsible for the bandwidth drop of ingress traffic. To make sure that this drop is caused by fragmentation and

not by wrong TC configurations, we run a second test, where we set the datagram size to the size of the MTU. If the hypothesis is valid and the bandwidth drop is caused by fragmentation, then we should see that the reached bandwidth is what we expect, namely the minimum of the bandwidth limit and the target bandwidth.

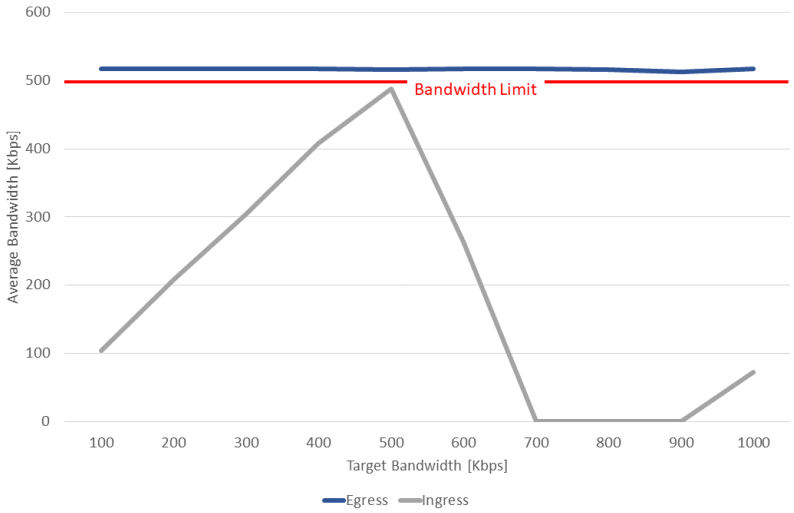

Figure 6.6: Evaluation of the Bandwidth with a Datagram Size of 10000 Bytes

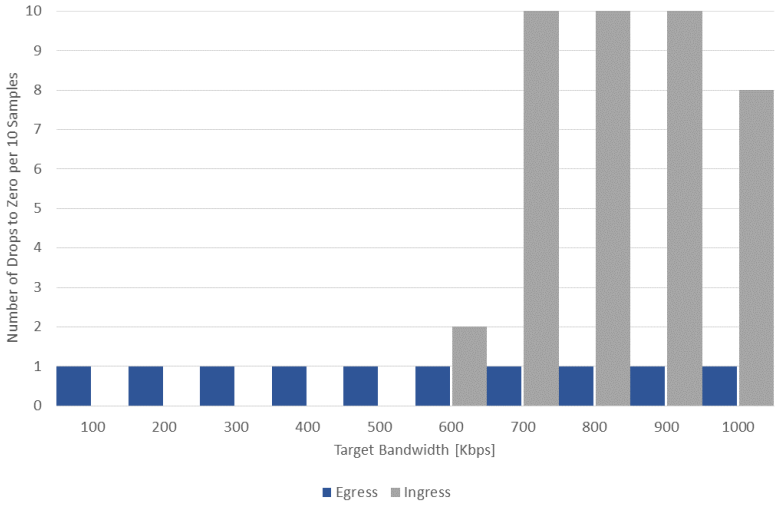

Figure 6.7: Evaluation of Bandwidth Drops with a Datagram Size of 100000 Bytes

For the second test, we set the datagram size to 1500 bytes, which is the size of the MTU. As visible in figure 6.8, both ingress as well as egress traffic first reach the target bandwidth, for as long as it is below the enforced bandwidth limit, and then it stabilizes approximately around the bandwidth limit. As expected, egress traffic stabilizes slightly above the enforced limit, because we allow short bursts at a higher bandwidth. Ingress traffic stabilizes as well, but also as expected slightly below the enforced bandwidth limit, because there is a certain overhead caused by redirecting ingress traffic to a virtual interface.
Therefore, we can conclude that the previously discovered strange behaviour of ingress traffic is not a side effect of the *scionlab_bw_limiter* but instead is normal behaviour that is caused by fragmentation (see section 6.3.4 for further clarification). Since our goal was to limit the bandwidth the same way a slow link would, we have to accept this behaviour and can therefore say that the bandwidth limitation was enforced successfully.

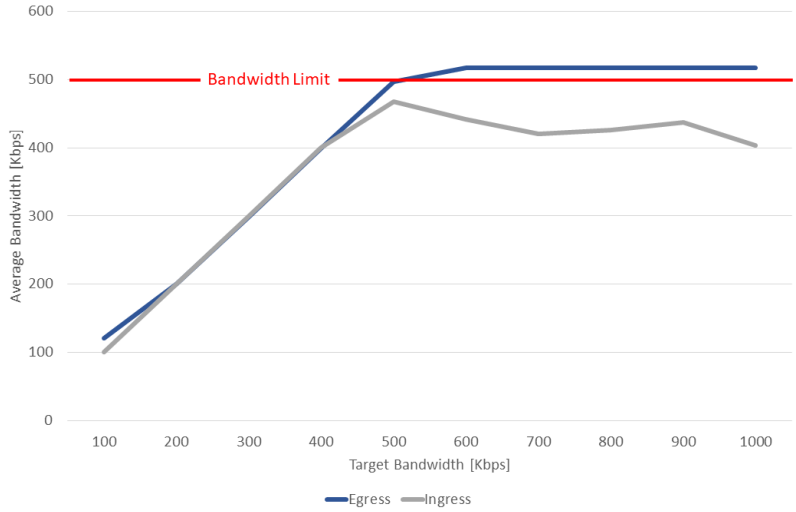

Figure 6.8: Evaluation of the Bandwidth with a Datagram Size of 1500 Bytes (Size of the MTU)

6.3.4 Fragmentation shown using Wireshark

Wireshark is a state of the art network traffic analyser. It is open source and its source code can be found on github.com[9]. We use wireshark to visualize ingress traffic that has been generated by iPerf3 and goes from our test server to the test client. Listing 6.5 and 6.6 show the iPerf3 commands used to generate the traffic, which is visualized in figure 6.9 and 6.10 respectively. In the first test (listing 6.5), we set the datagram size to 1000 bytes, which is smaller than the size of the MTU (which is 1500 bytes). In this case, we observe that no fragmentation of the test traffic occurs. Therefore, as visible in figure 6.9, each arriving IP-frame results in a valid UDP-datagram.
However, if we set the datagram size to 10000 bytes (listing 6.6), then a UDP-datagram needs to be fragmented into $\lceil \frac{10000}{1500} \rceil = 7$ IP-frames. As visible in figure 6.10, the first couple of frames get successfully reassembled into a valid datagram, where as at some point, here at packet number 48, the limitations start taking effect and no longer every IP-frame gets received. As visible in the right most column of figure 6.10, UDP-datagrams can no longer be successfully reassembled, which leads to many incomplete and therefore invalid datagrams.

```
1 iperf3 -c 192.168.17.129 -u -b 2Mbit -l 1000 -R
```

Listing 6.5: Example, where no Fragmentation occurs

No.	Time	Source	Destination	Protoco	Length	Info
30	0.146552970	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
31	0.146646943	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
32	0.146760198	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
33	0.146886843	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
34	0.146968851	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
35	0.147095632	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
36	0.147148683	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
37	0.147251198	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
38	0.147269477	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
39	0.147454611	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000
40	0.147530509	192.168.17.129	192.168.17.128	UDP	1042	5201 → 54923 Len=1000

Figure 6.9: Wireshark Output, where no Fragmentation occurs

```
1 iperf3 -c 192.168.17.129 -u -b 1Gbit -l 10000 -R
```

Listing 6.6: Example, where Fragmentation occurs

No.	Time	Source	Destination	Protocol	Length	Info
27	4.427097862	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=c716) [Reassembled in #33]
28	4.451329548	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=c716) [Reassembled in #33]
29	4.475549132	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=c716) [Reassembled in #33]
30	4.499782944	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=4440, ID=c716) [Reassembled in #33]
31	4.524009880	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=5920, ID=c716) [Reassembled in #33]
32	4.548272783	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=7400, ID=c716) [Reassembled in #33]
33	4.572446381	192.168.17.129	192.168.17.128	UDP	1162	5201 → 43926 Len=10000
34	4.591071242	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=c717) [Reassembled in #40]
35	4.615305951	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=c717) [Reassembled in #40]
36	4.639541069	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=c717) [Reassembled in #40]
37	4.663747994	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=4440, ID=c717) [Reassembled in #40]
38	4.687984807	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=5920, ID=c717) [Reassembled in #40]
39	4.712159594	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=7400, ID=c717) [Reassembled in #40]
40	4.736437388	192.168.17.129	192.168.17.128	UDP	1162	5201 → 43926 Len=10000
41	4.755023014	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=c718) [Reassembled in #47]
42	4.779243361	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=c718) [Reassembled in #47]
43	4.803455070	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=c718) [Reassembled in #47]
44	4.827646787	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=4440, ID=c718) [Reassembled in #47]
45	4.851873888	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=5920, ID=c718) [Reassembled in #47]
46	4.876108121	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=7400, ID=c718) [Reassembled in #47]
47	4.900395603	192.168.17.129	192.168.17.128	UDP	1162	5201 → 43926 Len=10000
48	4.918934675	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=c719)
49	4.943168987	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=c719)
50	4.967359119	192.168.17.129	192.168.17.128	IPv4	1162	Fragmented IP protocol (proto=UDP 17, off=8880, ID=c75b)
51	4.985961215	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=c7bc)
52	5.010180028	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=c7c1)
53	5.034397667	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=c842)
54	5.058626244	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=c8c7)
55	5.082872299	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=c94d)
56	5.107070172	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=c9d2)
57	5.131410428	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=ca4f)
58	5.155515065	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=cadf)
59	5.179745246	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=cb48)
60	5.204007846	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=cbcd)
61	5.228242043	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=cc4a)
62	5.252446073	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=7400, ID=cccd)
63	5.276634485	192.168.17.129	192.168.17.128	IPv4	1162	Fragmented IP protocol (proto=UDP 17, off=8880, ID=cd50)
64	5.295251868	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=cdd9)
65	5.319492603	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=5920, ID=ce61)
66	5.343713870	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=5920, ID=cec7)
67	5.367937840	192.168.17.129	192.168.17.128	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=5920, ID=cf4b)

Figure 6.10: Wireshark Output, where Fragmentation occurs

Chapter 7

Conclusion

7.1 Front End

Implementing the front end was the easiest part of the entire project. For that, I stuck mostly to the current practices and implemented the mechanisms that were additionally used for this project in a manner that is very similar to the mechanisms that were already in place. The intention of this approach was to make maintainability as easy as possible, especially since the SCIONLab server is still in development. I also made sure that the code that I contributed was well documented and easy to read.

7.2 Back End

Designing and implementing the *scionlab_bw_limiter* was definitely the part of the project that required the most research. Understanding and configuring TC turned out to be more challenging than expected. Furthermore it is rather unfortunate that there isn't really a reasonable API that lets us configure TC using Python. Therefore I had to write my own wrapper for TC. But I think that the implementation of the *scionlab_bw_limiter* works fairly well and is also easy to understand. Even though the results the *scionlab_bw_limiter* is delivering are not impeccable, they are still satisfying. Especially by considering the fact that, as far as I know, no better performing tool exists. In particular the results for ingress traffic were at first rather troubling. But after revealing the cause behind those strange results, I consider my results to be rather satisfying. The fact that the well established bandwidth limiter *wondershaper* shows similar results, confirms my findings.

7.3 Difficulties that Arose

The biggest challenge I was facing, was enforcing a bandwidth limit on ingress traffic. TC was mainly designed to do traffic shaping in order to improve the quality of service. For improving the quality of service, egress traffic is much more relevant than ingress traffic. Therefore TC is not as advanced, when it comes to handling ingress traffic, as I wished it would be.
Furthermore, testing turned out to be quite difficult as well. It's not easy to create a realistic test environment, which is still simple enough such that test results can be interpreted in a meaningful way. If test results turn out differently than expected, it is often not clear whether the measurement was wrong, the test configuration was faulty, the test environment was unrealistic or the implementation was buggy. It might as well just be that the testing tool is limited in its measurement capabilities.

7.4 Lessons Learned

This bachelor thesis project offered me the opportunity to gain a deep understanding of how IP traffic can be managed in order to enforce a bandwidth limit and what possible complications and consequences might occur. It also provided me with a holistic view over SCION, SCIONLab and in general how a novel internet architecture can be tested in a distributed testbed. Furthermore, I acquired a better understanding of how Linux utilities that are used for networking purposes work. Finally, this project emphasized the importance of extensive testing and showed me how difficult it is to have an accurate yet easy to use testing tool.

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
ESnet	Energy Sciences Network
ETH	Eidgenössische Technische Hochschule (Swiss Federal Institute of Technology)
HTB	Hierarchy Token Bucket
IFB	Intermediate Functional Block
IP	Internet Protocol
ISD	Isolation Domain
ISP	Internet Service Provider
JSON	JavaScript Object Notation
MTU	Maximum Transmission Unit
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
TCP	Transmission Control Protocol
TRC	Trust Root Configuration
UDP	User Datagram Protocol
UML	Unified Modeling Language
VM	Virtual Machine

Bibliography

[1] Network Security Group ETH Zurich. *SCION Internet Architecture*. 2019. URL: https://www.scion-architecture.net/ (visited on 06/17/2019).

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

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

[4] Manuel Meinen, Matthias Frei, et al. *scionlab Branch: bw-limit*. 2019. URL: https://github.com/ManuelMeinen/scionlab/tree/bw-limit (visited on 08/18/2019).

[5] Manuel Meinen. *SCIONLab_Bandwidth_Limiter*. 2019. URL: https://github.com/ManuelMeinen/SCIONLab_Bandwidth_Limiter (visited on 08/18/2019).

[6] Bruce A. Mah, Jon Dugan, Todd C. Miller, et al. *iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool*. 2018. URL: https://github.com/esnet/iperf (visited on 06/17/2019).

[7] Bruce A. Mah et al. *strange mismatch behavior UDP OUT OF ORDER #457*. 2016. URL: https://github.com/esnet/iperf/issues/457 (visited on 06/14/2019).

[8] Bert Hubert, Jacco Geul, and Simon Séhier. *wondershaper*. 2002. URL: https://github.com/magnific0/wondershaper (visited on 06/17/2019).

[9] Gerald Combs et al. *wireshark*. 2019. URL: https://github.com/wireshark/wireshark (visited on 08/18/2019).

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):

Bandwidth Limit Enforcement for SCIONLab

Authored by (in block letters):

For papers written by groups the names of all authors are required.

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

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)**
Zurich, 18.08.2019	

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