Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Bachelor's Project

Framework for network management to support simulation of varying network conditions

Petr Praus

Supervisor: Ing. Tomáš Černý M.S.C.S.

Study Programme: Softwarové technologie a management, Bakalářský

Field of Study: Softwarové inženýrství

May 19, 2011

Aknowledgements

I would like to voice my great appreciation to Ing. Tomáš Černý M.S.C.S. for guiding me through such a complex assignment and my team members Slavka Jaromerska and Lubos Matl for their insightful remarks and enjoyable cooperation.

Declaration

Abstract

Modern peer-to-peer systems are increasingly complex yet their testing still requires controlled and repeatable environment. The complexity of these systems and their protocols severely lowers feasibility of simulation in this regard. The solution that permits the use of the actual applications and associated protocols in the controlled and repeatable environment is emulation. Emulation allows real hosts to communicate through a transparent virtual network modelling the real network. However, the virtual network is often created by the emulation layer placed in the operating system's kernel posing a certain challenges regarding practical usability – the features needed for creating the environment are there but a significant expertise is required to properly use them. In addition, the components in the kernel are often primarily concerned with practical traffic shaping and Quality of Service in general rather than emulation. This work presents a framework significantly simplifying the creation and configuration of the network emulation environment while pointing out and helping avoid some common pitfalls an unsuspecting user might run into. Those wishing for more in-depth look on the possibilities for network emulation on Linux are presented with an extensive theoretical background and host of related work.

Abstrakt

Moderní peer-to-peer systémy jsou stále složitější, ale jejich testování stále vyžaduje kontrolovatelné a opakovatelné prostředí. Složitost těchto systémů a jejich protokolů významně snižuje praktickou použitelnost simulace pro jejich testování. Řešení dovolující použití reálných aplikací a souvisejících protokolů je emulace. Emulace umožňuje reálným hostům komunikace přes pro ně transparentní virtuální síť modelující reálnou síť. Virtuální síť je často vytvářena emulační vrstvou umístěnou v jádře operačního systému. To představuje nezanedbatelnou složitost zejména v oblasti praktické použitelnosti, protože pochopení a efektivní využití funkcí jádra vyžaduje relativně rozsáhlé znalosti. Komponenty jádra jsou navíc často určeny spíše pro praktické omezování provozu za účelem zajištění kvality služby (Quality of Service) než emulaci. Tato práce představuje framework podstatně zjednodušující tvorbu a konfiguraci síťového emulačního prostředí a zároveň upozorňuje na některé záludnosti, které mohou překvapit. Pro čtenáře vyžadující hlubší pohled do možností emulace sítí v Linuxu, práce obsahuje rozsáhlou teoretickou kapitolu a velké množství souvisejících zdrojů.

Contents

List of Figures

List of listings

Chapter 1

Introduction

Motivation for this work stemmed from the need for efficient network emulation for Cooperative Web Cache project (briefly described later in section 1.1 or [1, 2]). We needed a way to test our implementation on a larger network of at least tens but preferably hundreds of network nodes. Our resources were very limited and we needed the network emulation to work in the scope of a very few real machines where one physical machine would represent several Cooperative Web Cache nodes. Fortunately, requirements of a single CWC node were relatively slim and we soon found out it is possible to run about forty CWC nodes on a dual-processor machine with 4 gigabytes of memory without hitting swap or incurring CPU load that would skew our measurements. This approach looked feasible and we set about finding the best way to emulate a real network environment on a Unix-like system in the scope of one machine. Network emulator as a separate device functioning as a switch/router (L2/3) for a network segment seems to be a traditional approach mainly by the commercial sector keen on selling “the box”. Admittedly, the box can be quite clever but is always immensely expensive and still unable to meet the need for the aforementioned local emulation where one physical machine runs several application instances.

Both Linux and FreeBSD have their own IP bandwidth management and network emulation solutions with features often comparable to those of commercial offerings. FreeBSD has dummynet [3] with ipfw. Dummynet was originally developed for testing network protocols but since grown to a general bandwidth management solution. It still retains its roots and offers a good feature set in terms of network emulation. Bandwidth management and quality of service in Linux is done by a Traffic Control module [4, 5]. It is not a module per se but rather a set of functionality in the kernel controlled by the user-space utility *tc* from iproute2 package. We eventually choose the Linux solution because we are by far more adept at GNU/Linux administration than *BSD systems.

Alas, the aforementioned solution has a few problems because we are trying to use it in a quite nontraditional setup. For a start we had to choose a unique identifier for each instance of our application (CWC node in our specific case). Initially we tried to use a set of ports but that proved to be a bad idea. It forces the application instances to use artificially made up ports and more importantly know in advance what ports will the application use. This is not a big deal for listening ports but application very rarely specify a source port when making a connection making it next to impossible to shape or police such traffic based on ports because it is too late to create a traffic shaping rule at the moment a connection is established and a source port is already assigned by the operating system. IP address is more suitable candidate because it is more static. However, it has a similar problem (best-guess assignment of the source identifier at the operating system's discretion) but only when dealing with local traffic. If an application omits an IP address when calling connect() or bind() libc functions, operating system assigns IP address it thinks is the closest. And what closest address there can be other than the same one. This results in a traffic originating and terminating at the same IP and since the traffic is accounted by IP a client peer leeches the bandwidth of a server peer.

The possible solution is a lightweight virtualization such as BSD jails, if we were using BSD. But Linux lacks a practical lightweight virtualization with sufficiently independent networking and hassle-free management (the likes of OpenVZ certainly do not fall into this category). In addition, any form of virtualization would necessary incur overhead decreasing the number of nodes we could start on one machine – a crucial parameter. Even more lightweight solution that we actually used is overriding the aforementioned (g)libc functions connect() and bind() through dynamic linking with custom library placed in `LD_PRELOAD` [6] environment variable to prevent operating system's IP stack from automatically assigning an IP address. A related problem is that all local-to-local traffic goes through the loopback interface no matter what the address. It is logical and normally highly desirable behavior but if you are trying to make all application instances equal across multiple machines you have to aggregate traffic from the network interface (traffic from instances running on other machines) and the loopback interface (traffic from local instances) to correctly measure and shape or police traffic. The solution to this problem is described in section 2.4.

This work considers emulation purely from network point of view – it separates the application instances with regard to networking and completely disregards sharing of other resources. For example, if one application instance begins a computationally intensive task it will most probably have detrimental effects on other nodes and their performance. It is necessary sacrifice for running multiple instances on the same machine. Again, virtualization could be possibly used to partially avoid this effect but it makes system substantially more complex and incurs significant overhead. On the other hand, our instances were relatively lightweight without the need for computationally intensive operations and our machines were multi-core thus reducing probability that the nodes would deprive each other of resources at the critical moment.

1.1 Cooperative Web Cache

Technology for web service distribution does not scale naturally with increasing demand resulting in sharp rise of operational costs in case of a successful venture. These costs are manageable if you are a large, established for-profit provider because your revenue tends to scale if you are doing it right. But what if you do not have any revenue either because you are a start-up company or non-profit organization such as Wikimedia or EFF? You need to offset your costs by leveraging your users. Wikimedia foundation spends more than 1 million in US dollars annually for Internet hosting. Cooperative Web Cache project promises to scale web service delivery and cope with demand spikes more naturally by employing clients in content replication. Indeed, projects like BOINC demonstrated many would be more than willing to donate their networking and computational resources to help with non-profit project's hosting.

Cooperative Web Cache was hitherto presented in two papers [1, 2]. [1] lays down some theoretical foundation regarding peer-to-peer networks and distributed hash tables and presents an extensive case study proving general validity of our approach. [2] directly builds on the first paper and its case study shows new techniques for improving upon the previous results while proposing several future directions our work could go. The Cooperative Web Cache itself was mostly implemented by Lubos Matl [7], test design and a testing framework was programmed by Slavka Jaromerska [8]. The testing framework in turn used my work for the actual network emulation.

1.2 Goals

This work aims to convey experience in network emulation of complex P2P applications using Linux Traffic Control while serving as an intelligible guide to the technology. The inseparable goal of the project is object-oriented pure-Python (without any C extensions or wrappers) framework called *ShaPy Framework* for accessing traffic control capabilities of the Linux kernel using the Netlink interface (where possible). The framework is used in *Shapy Emulation* – an extension that makes construction of virtual networks for emulation a breeze. Python is becoming the language of choice for Linux system administration and offers seamless integration with BSD sockets essential for a Netlink communication making Python a natural choice for Linux traffic control configuration. The standard way of controlling traffic control capabilities is user-space configuration tool called *tc* which is part of *iproute2* suite used for configuring almost every aspect of Linux networking. The problem with programmatic control of tc is you have to construct plain text commands and run them in a shell. Therefore *tc*'s primary application lies in manual introspection and configuration or shell scripting at best – much like *iptables* tool. Absence of any sensible framework complicates not only control but also a retrieval of existing settings and statistics.

1.3 Related work

In terms of *Netlink/Traffic Control frameworks* three related projects exist. The first is *Linux QoS Library* [9]. It is written in C with GObject and therefore using it with Python would require writing substantial amounts of GObject wrappers, generating boilerplate code and subsequently compiling it just to send a few messages via Netlink. Also, LQL is not maintained since early 2005. The second is *libnl* [10]. Libnl is actively developed, has sensible API but unfortunately it is written in C which is unsuitable for our Python/Java environment. The rest of the CWC testing framework is written in pure Python and LQL/libnl approach would pose integration problems. The third project is *Traffic Control - NextGeneration* [11]. This time an entirely new C-like language was conceived for the sole purpose of controlling QoS facilities of Linux kernel. No doubt its superb tool but one which very poorly integrates into a Python ecosystem. TCNG has some very neat features, for example it can compile a filter as a kernel module resulting in a greatly increased matching speed. In addition, ShaPy aims to eventually provide more functionality than just traffic control. Specifically, it might become a generic Python library for all network-related configuration (for example routing).

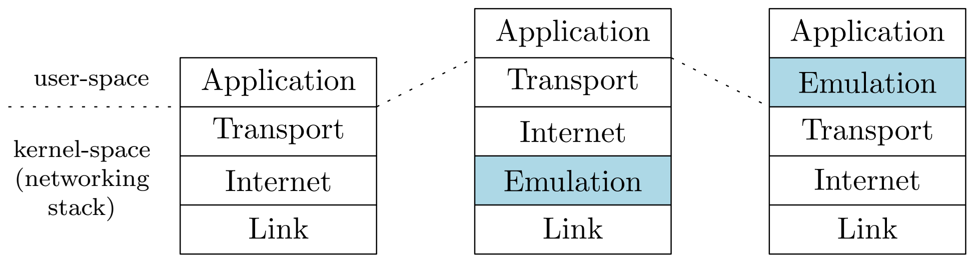

Figure 1.1: Emulation layer placement in the networking stack

The following related work is focused on the more abstract emulation goal rather than just being a Netlink framework. There are several different approaches how to provide a controlled and repeatable environment while allowing use of real protocols and applications. The environment is a virtual network modelling some predefined behavior. Real hosts are then connected through the virtual network to each other. The first approach is creating an *emulation layer* inserted into the protocol stack. The layer intercepts the traffic and applies the prescribed behavior on the packets. Practical implementation of the emulation layer differs quite a low. It can be inserted in the *kernel-space* (the approach this work takes inserts emulation layer below transport layer) or *user-space*. The second approach is emulating virtual network through the use of real-time network simulator. This approach is out of scope of this work and will not be discussed here. You can refer to the paper *Advances in Network Emulation* [12] for further details. This paper is also the source of the aforementioned network emulator classification and provides good overview of network emulation software. Figure 1.1 depicts possible placements of the Emulation layer within the traditional 4-layer networking model specified by the RFC 1122 [13].

Kernel-space emulation layer is used in dummynet [3], NIST Net [14], WANem [15] and WAN Emulator which was used in evaluation of TCP Vegas algorithm [16]. WANem is actually built from very similar components as the ShaPy, namely Netem and HTB, but lacks the local emulation. MicroGrid [17] uses user-space method and captures system calls through dynamic linking and *LD_PRELOAD* ([6] describes how this mechanism works) and connects to the environment in this way. This method is obviously much harder to implement, it is prone to leaking calls and most importantly fails to emulate how *real* OS stacks will behave towards the employed network protocols. REAL 5.0 [18] "for studying the dynamic behavior of flow and congestion control schemes in packet-switched data networks" uses somewhat similar approach but my understanding is that it requires even deeper alteration of tested program and dynamic linking is not enough. I will skip the commercial hardware solutions for now, they are briefly discussed in section 2.5. All mentioned projects have in common that none of them (commercial offerings including) offer local emulation within the scope of one operating system user-space. Our approach seems to be hitherto unique.

Chapter 2

Theoretical Background

This chapter will explain the theoretical background necessary to understand underlying concepts behind the framework presented in chapter 3. Section 2.1 introduces inter-process communication protocol Netlink and its model, section 2.2 follows up with the discussion of practical implementation of the previously introduced Netlink model in Linux. Section 2.3 delves into the Linux Traffic Control, explaining what is a queueing discipline, a token bucket or a link-sharing goal. The chapter concludes with two emulation-oriented specialty modules – Intermediate Function Block, basically a virtual interface and NetEm, a special queueing discipline capable of emulating various conditions commonly occurring on real-world networks.

2.1 Introduction to Linux Netlink

Netlink is a protocol for inter-process communication, it allows for extremely flexible communication between a set of processes in kernel-space or user-space. The flexibility comes at a price – Netlink is a very low-level protocol and the user is entirely in control of all aspects of packet creation, including header construction or datatype alignment. A Netlink message is addressed by an inherently local process ID (PID) and thus cannot traverse host boundaries[1]. Low-level nature of Netlink combined with relatively sparse documentation of various implementation details scattered around kernel header files and various man pages prevents using Netlink as-is without writing at least a basic library. For example, creating a HTB or a CBQ class by a Netlink message requires close examination of `include/linux/pkt_sched.h`. That is not to say the Netlink is poorly documented – for someone apt at kernel development the aforementioned documentation in the form of kernel header files is probably more than sufficient. But learning curve is quite steep and if your goal is something different and Linux traffic control is just another tool you might find it quite intimidating. On the other hand, some working knowledge of how Linux traffic control works is handy – especially for using ShaPy Framework. Netlink is “standardized” in informational RFC 3549 [19].

[1]This does not mean you cannot use Netlink remotely, you just need to use additional protocol (such as XML-RPC) for transmitting the commands or entire Netlink messages between hosts.

2.1.1 Netlink concepts

Since this work is primarily concerned with networks, the primary focus with Netlink will also be on networking and QoS in particular. Netlink protocol can be seen as a communication protocol between a *Control Plane (CP)* and a *Forwarding Engine (FE)* (as defined in the ForCES charter [20]). A Control Plane is an execution environment containing subcomponents (Control Plane Components, CPC) which control IP services executed by a Forwarding Engine through its FECs (Forwarding Engine Components). The relationship between CP and FE is that of a master and slave. "In essence, the cohesion between a CP component and an FE component is the service abstraction." (RFC 3549). *Control Plane Components encompass signalling protocols* with the ultimate goal of configuring the FE (the second Network Element (NE)[2], the first is CP) and its FECs. FEC treats the packet with the goal of achieving IP service. Configuration of each individual FEC ultimately controls the fate of the packets traversing the FE.

2.2 IP Forwarding Engine in Linux

A packet first physically arrives at a network interface where it is decoded, sent into the link layer, stripped of a frame layer and finally into the Internet layer (as defined in RFC 1122 [13]) where the figure 2.1 starts. The figure depicts the implementation of a Forwarding Engine in Linux. The only mandatory component from a standards point of view is *Forwarding* – a component that takes care of the Router requirements mandated by RFC 1812 [22]. First, the kernel calls a Netfilter [23] hook and the packet goes through the `PREROUTING` chain. Then it is ushered into the Ingress Traffic Control module where the metering and policing occurs. This module only allows classless qdiscs and packets can only be dropped (when they exceed a set rate) or forwarded to a different interface but never delayed. This is where Intermediate Functional Block [24] comes handy but let us not digress and suppose the packet emerged from Ingress Traffic Control unharmed. It continues to the aforementioned Forward module where the *routing decision* occurs. The kernel looks at the IP header and decides whether the packet is destined for the local host. If yes, a Netfilter hook is called again, this time the packet travels through the ip_tables[3] `INPUT` chain, it is delivered to the local IP stack, the socket opened by an application and finally to the application process itself. If the packet is not destined for the machine acting as a forwarding engine (FE), it is forwarded to the next hop determined by the routing rules. The packet then travels through two additional ip_tables chains – `FORWARD` and `POSTROUTING` and at last arrives at the Egress Traffic Control. This is the place where the entire range of shaping options can be applied. A locally generated packet travels from a socket through the `OUTPUT` chain and then follows the same route as the forwarded one.

[2]An IP network element (NE) is composed of a numerous logically separate entities that cooperate to provide a given functionality (such as routing or IP switching) and yet appear as a normal integrated network element to external entities. Two primary types of network element components exist: control-plane components and forwarding-plane components. (RFC 3654 [21])

[3]There is substantial difference between ip_tables and iptables. The former is a kernel module that makes use of Netfilter hooks while the latter is a user-space configuration utility for the former (the kernel module).

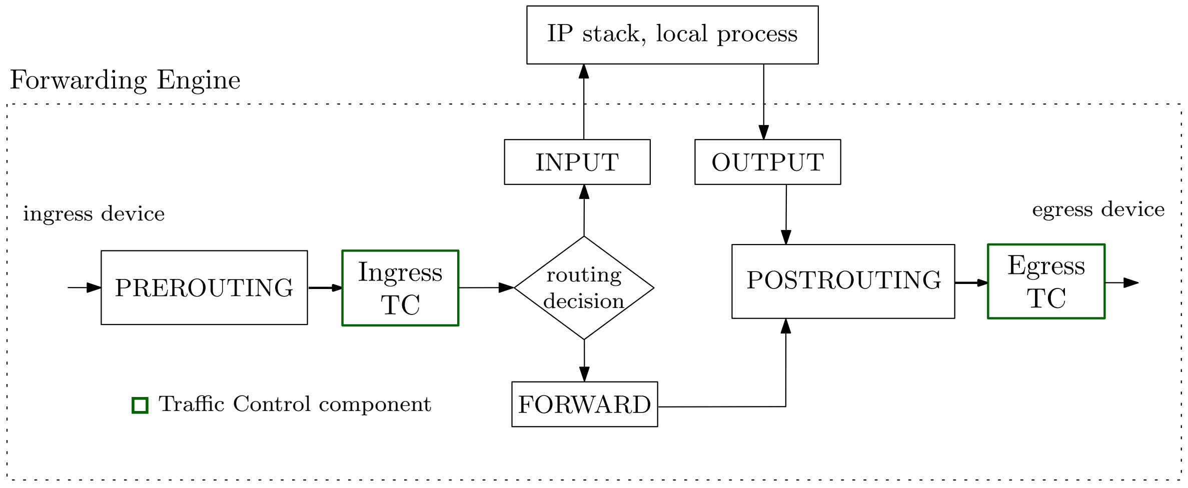

Figure 2.1: Linux forwarding engine

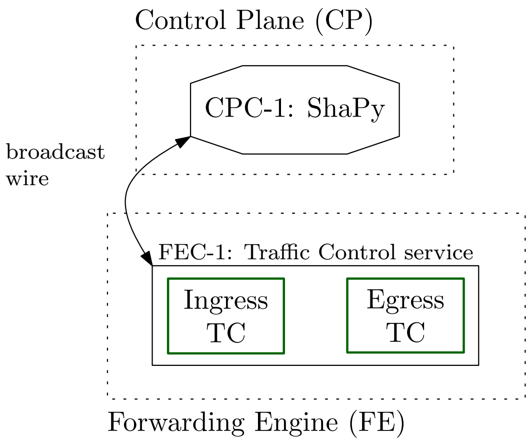

Figure 2.2: Relationship between the CP and the FE in context of ShaPy

2.2.1 Netlink architecture

The FEC needs an information from the CPC on how to operate a service and CPC in turn needs the status information from the "field", to make a proper decision. Obviously a communication protocol defining the syntax and semantics of messages transmitted on the (figurative) wire is needed. The protocol is provided by the *Netlink service*. It is a mediator capable of transmitting messages between user-space and kernel-space. It may check the adherence of the message to the protocol (Linux Netlink implementation does the check). In Linux the mediator service interface is realized by a socket in raw mode. This implies the protocol is unreliable. However, the peers may choose to define a reliable protocol using sequence numbers and ACK messages. RFC 3549 summarizes: "Netlink allows participation in IP services by both service components."

Figure 2.2 shows the relationship between ShaPy (the Python framework for Linux Traffic

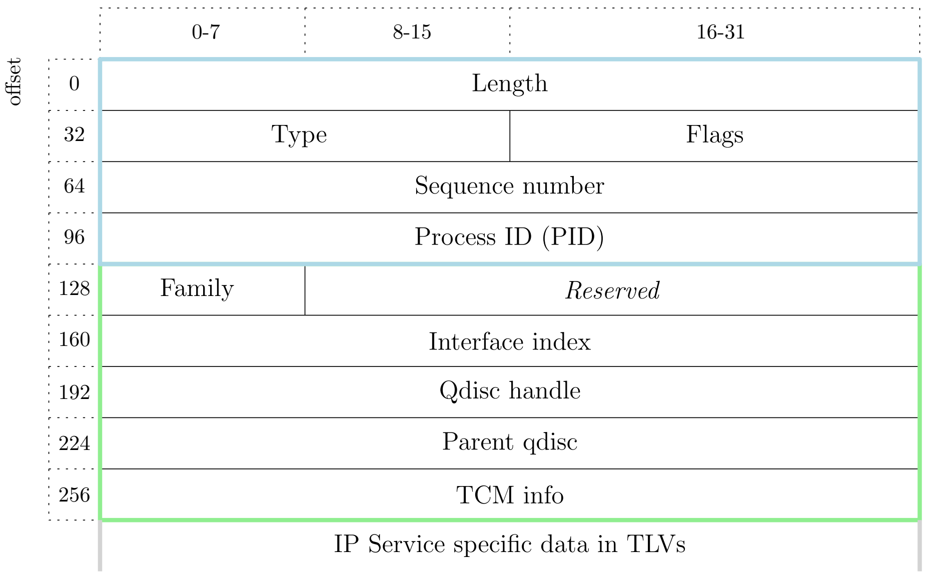

Figure 2.3: Netlink message with Traffic Control service template

Control defined in Goals, section 1.2) as a CPC, and the Traffic Control service as a FEC. FECs and CPCs are modelled as nodes connected through a broadcast wire. The wire is service-specific and thus separate for routing IP service (`NETLINK_ROUTE` alias rtnetlink, Traffic Control falls into this category), firewall service (`NETLINK_FIREWALL`) or ARP service (`NETLINK_ARPD`). A node connects to the specific wire and registers itself with the mediator which allows the node to receive or emit broadcast, unicast and even multicast messages on the wire.

2.2.2 Netlink protocol

Netlink messages are used for communication between FECs and CPCs. A message can be used for configuring a FEC (its behavior toward the passing packets), retrieving statistics or event notification. A Netlink message has three parts (see figure 2.3): general message header (blue), IP service template (green) and arbitrary number of TLV (type-length-value) elements for IP service attributes (CPC $\rightarrow$ FEC). Direction FEC $\rightarrow$ CPC instead typically contains a Netlink ACK message. For format of the Netlink ACK message refer to section 2.3.2.2 of RFC 3549. General Netlink message format (the first part) is fixed-length and universal across all services. The service template (the second part) contains general information typical for a given service – such as what network interface should be configured as a result of this service request. The attributes (stored in TLVs) are specific to the desired action (for example a request to create an IP address on the interface specific in service template). Figure 2.3 shows a Netlink message format with Traffic Control service header and outlined TLVs.

On a practical note, CPC gains access to the Netlink service by creating a raw socket (`SOCK_RAW`) with address family `AF_NETLINK`. The socket's protocol (last argument to the *socket()* call) defines desired IP service (such as `NETLINK_ROUTE`, see netlink(7) man page for complete list). Listing 2.1 shows how an rtnetlink socket could be created in Python (its practically the same as in C, because Python's `socket` module moreorless encapsulates the standard C socket API). After a connection is established, a CPC can subscribe to service's events, issue a service-specific configuration command, or a statistics-gathering command. RFC 3549, rtnetlink(7) and netlink(7) contain exhaustive list of possible commands (service templates) as well as explanation of the fields in the general Netlink header. Section 3.5 contains more details about (rt)netlink socket initialization.

Listing 2.1: Creating an rtnetlink socket

```
rtnetlink_socket = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
```

2.3 Linux Traffic Control

This section will describe in greater detail a plethora of Linux kernel IP bandwidth management options. First it is important to define a few terms. *Egress* traffic is outgoing, *ingress* traffic is incoming. *Shaping* usually entails delaying packets in some queue to avoid link congestion and subsequent traffic buffering on other network elements outside our control. Retaining control over the queue allows us to decide which packets will get delayed (dequeued later) or dropped entirely. This *typically* makes sence only for egress traffic. Managing ingress traffic is somewhat trickier task as you generally do not have control over the other end of connection. It is tricky precisely because the aforementioned control over the queue. As you cannot directly control what others send to your interface you can only *police* such traffic – limit the rate at which you accept it and drop (or reclassify) the rest. In case of TCP, its flow control mechanisms result in slowing down traffic. In case of UDP, such endeavor is generally doomed as dropping UDP packets does not result in sender throttling (unless an upper layer protocol detects the missing packets and resends). In a traditional sense of quality of service it does not make sense to delay packets that already arrived – hence the lack of classful qdisc support for ingress. However, it does make sense for an emulation environment. How to use entire shaping potential even for ingress is described in section 2.4 which discusses virtual network device called Intermediate Functional Block [24].

2.3.1 Queues, queueing disciplines, tokens and buckets

Every interface has to have a queue where the packets waiting to be transmitted can be stored. The nature of queue along with the queueing discipline (qdisc) determines the fate of the packet travelling through. Whenever the kernel needs to send a packet through an interface, it is enqueued to the root qdisc configured for that interface. Immediately afterwards, the kernel tries to get as many packets as possible from the root qdisc for giving them to the network adaptor driver. A *queueing discipline* is basically a queue (packet) scheduler *encapsulating* the other two TC components – *classes and filters*. It can also be imagined as a logic behind the queue behavior. For example a FIFO scheduler ensures that packets are dequeued in exactly the same order as they were enqueued. There are two basic types of qdics, *classless* and *classful*. A qdisc is identified by a 32-bit number split into major:minor (16-bit) parts. Major part must be unique per-interface. Minor part is always zero for qdiscs with the exception of Ingress (`ffff:fff1`) because it is reserved for a

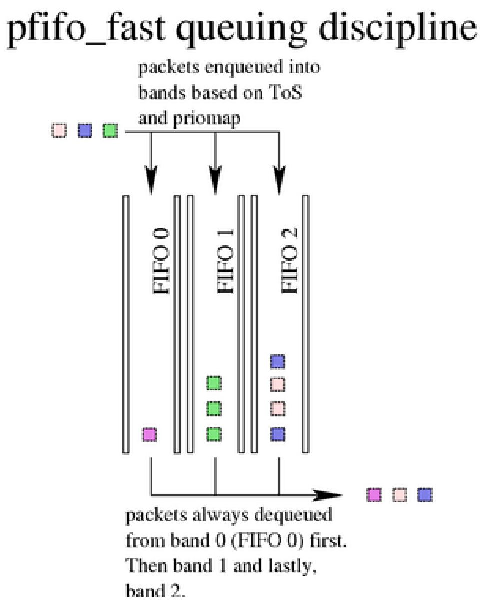
pfifo_fast queuing discipline
packets enqueued into bands based on ToS and priomap
FIFO 0
FIFO 1
FIFO 2
packets always dequeued from band 0 (FIFO 0) first. Then band 1 and lastly, band 2.

Figure 2.4: pfifo_fast schema

class identifier (see section 2.3.2). A network interface has always exactly one *root queueing discipline* (“1:0” or “1:”, zero can be omitted).

Classless queueing disciplines are straightforward, they function on their own without the need for any classification. A typical example of such qdisc is pfifo_fast with prioritization based on the IP packet’s ToS field meaning the packet with lower priority is dequeued only when there are no higher priority packets in the queue (see figure 2.4). This is the default qdisc every network interface gets from the start. The other example is Token Bucket Filter capable of limiting speed at which packets can traverse through. Choosing a classless qdisc as a root obviously excludes using any other qdiscs.

The following quotation is from the theoretical paper written by Sally Floyd and Van Jacobson, both well known for their work on TCP/IP stack in 1995. The paper was the first to propose a hierarchical structure for a traffic control. It explains why we actually need the hierarchy and later led to the implementation of classful qdiscs in Linux.

> The various requirements for link-sharing naturally lead to a requirement for hierarchical link-sharing. For example, the bandwidth on a link might be shared between multiple agencies, and each agency might want to share its allocated bandwidth between several traffic types. This leads to a *hierarchical link-sharing structure* associated with an individual link in the network, with each *class* in the link-sharing structure corresponding to some aggregation of traffic.
>
> – *Link-sharing and Resource Management Models for Packet Networks [25]*

Classful queueing disciplines are far more flexible. They contain filters which *classify* packets into *classes*. Each class is in turn managed by another (attached) queueing discipline.

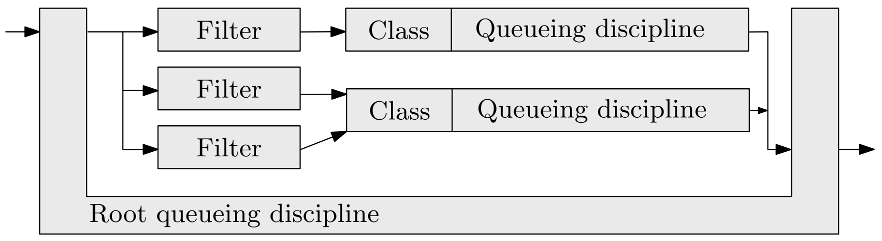

Figure 2.5: Generic layout of Linux Traffic Control components

This queueing discipline could be a trivial FIFO (implicit if you don't specify it) or another classful queueing discipline leading to a tree structure. You can see a very complex scenarios can be built this way. When the kernel tries to dequeue a packet from a root classful qdisc (only one such qdisc exists for any given network interface) it can come from any of the classes. This process is shown on figure 2.5. Hierarchical Token Buffer (HTB) or Class Based Queueing (CBQ) are prominent examples of classful queueing disciplines.

A concept of tokens and a bucket is at the basis of most modern queueing disciplines. Accurate measurement of dequeueing rate and burst handling would require relatively complex logic and the concept of tokens is a logical construct to simplify this problem. Tokens are generated at a steady rate corresponding to the desired traffic rate and dequeueing a packet spends a token[4]. When there is no token available a packet must wait in a queue until one is available. When there is no packet to consume the generated tokens, the tokens are instead stored in a bucket of limited capacity for future use. When a burst arrives the tokens in the bucket are spent without incurring unnecessary delay by waiting for the new tokens. Think of it as not wanting to restrict casual web user (bursty, once-in-a-while high-speed traffic) and instead wanting to shape large downloads (constant, high-speed traffic) causing excessive load on the network. *Leaky bucket* [26] is a very similar and directly transferable concept to the described *Token bucket*. *Token Buffer Filter* qdisc operates in exactly this fashion. Figure 2.6 (inspired by The Linux Documentation Project howto on TC) illustrates this mechanism.

2.3.2 Classes, Hierarchical Token Bucket

Classes are specific to classful qdiscs and are used to create a "behavior hierarchy" represented by a tree (see section 2.3.1 for more details on classful qdiscs). For example HTB classes (see section 2.3.2) with common parent share the parent's bandwidth ceiling. This section describes only HTB and leaves out CBQ because HTB goals are superset of CBQ thus satisfying them also satisfies CBQ goals. See the theoretical foundation paper for hierarchical link-sharing [25] (section 2) for detailed explanation.

A class is an aggregation for different types of traffic (e.g. ACK packets in TCP), service classes (such as audio, video, ...), organizations, or protocol families. It is identified by a 32-bit class ID structured like qdisc IDs (two 16-bit major:minor parts) with major number corresponding to the qdisc instance of a class and minor number identifying the class within

[4]In fact, real implementation is a little bit more complex as this approach does not account for varying packet size.

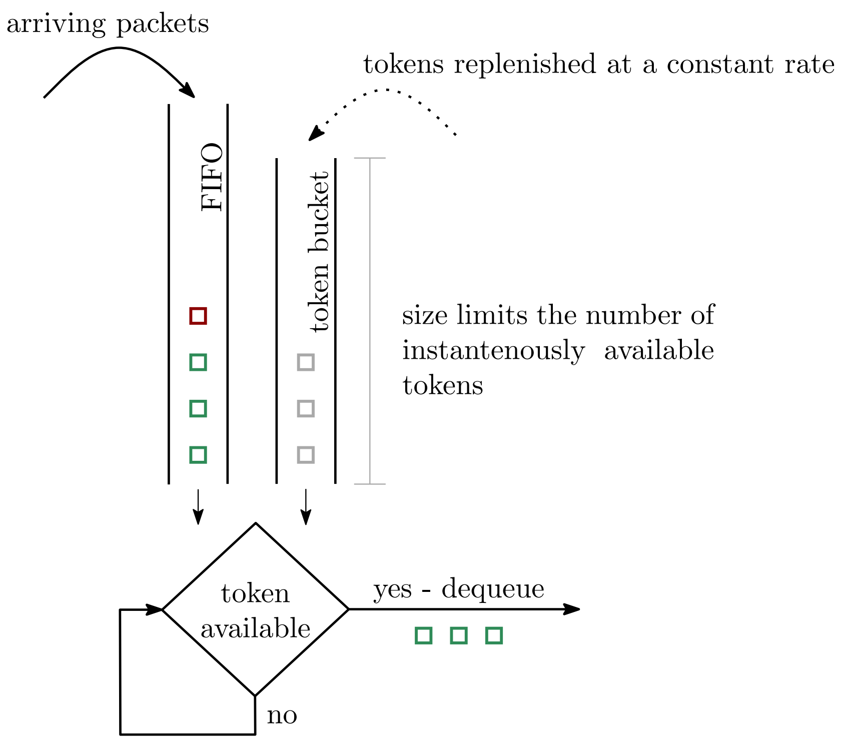
arriving packets
tokens replenished at a constant rate
FIFO
token bucket
size limits the number of instantenously available tokens
token available
yes - dequeue
no

Figure 2.6: Token Buffer Filter schema

that instance. The hierarchical link-sharing structure (as defined in section 2.3.1) specifies the desired policy in terms of the division of bandwidth for a particular link in times of congestion.

The *first link-sharing goal* HTB needs to satisfy is that each class with sufficient demand should be able to receive roughly its allocated bandwidth, over some interval of time. A *secondary link-sharing goal* is that when some class is not using its allocated bandwidth, the distribution of the extra bandwidth among the other classes should not be arbitrary, but should follow some appropriate set of rules. Each HTB class has therefore a guaranteed rate (allocated bandwidth), ceiling (how much traffic it can redistribute to its children in total), priority and quantum. A child class charges traffic going through to its parent class. Excess bandwidth is divided first according to priority – highest priority classes are served preferentially. Quantum is used for division on the same priority level. Inner classes only redistribute bandwidth to the leaves (classes without children) and only leaves can hold a packet queue. Logically, HTB is a hierarchy of Token Buffer Filter classless qdiscs but actual implementation has nothing to do with TBF.

Instead of diving deep into a theory[5] behind hierarchical link-sharing, let's have a practical example. We have a 1 Mbit link we want to sell three times over. We will have three service tiers with different aggregation ratio representing guaranteed proportion of theoretical maximum capacity (1:2 for the first customer, 3:8 for the second and 1:8 for the third)

[5]However, HTB author's site contains an excellent discussion of the implementation details behind HTB. See http://luxik.cdi.cz/ devik/qos/htb/manual/theory.htm

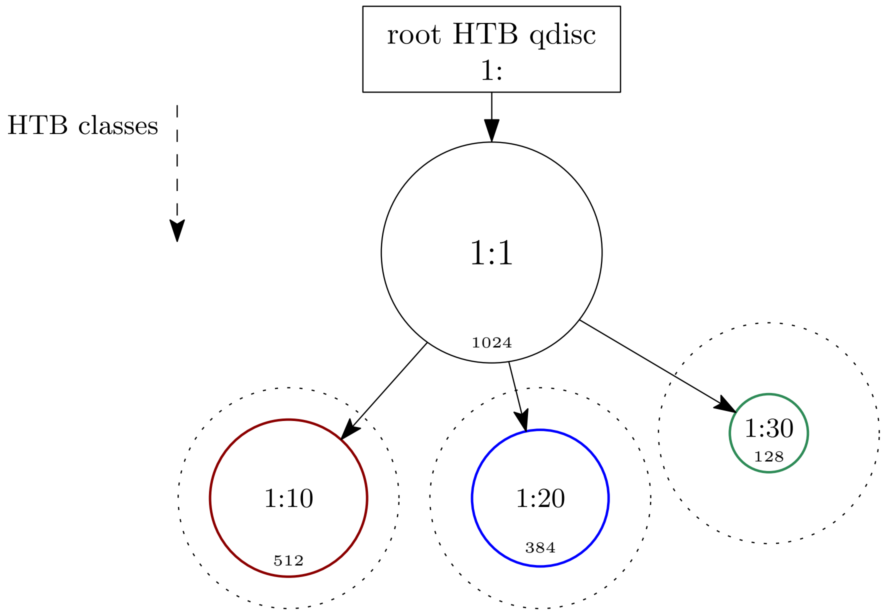

Figure 2.7: Hierarchical Token Bucket example

and sell each customer entire 1 Mbit. In terms of HTB, each customer's traffic will be represented by a class and our physical link will be class 1:1. Since we are honest provider we want to maintain the aggregation ratio and we will set guaranteed bandwidth (`rate`) on each class to the respective portion of bandwidth (512 kbps in case of the first customer, etc.). We also want to keep the promise that a customer's link speed should be at least sometimes 1 Mbit. We can achieve this by borrowing currently unused bandwidth from other classes (`ceil`) when one class has demand for more than its guaranteed rate. Figure 2.7 shows the described situation in graphical form and listing 2.2 as *tc* rules. The root qdisc also contains a parameter specifying that unclassified packets will be sent to class 1:30 (`default`). The figure and the listing are just different representations of the same, class colors are corresponding. The aforementioned figure and listing omit classifiers, they will be explained in the next section (2.3.3). Figure 2.9 shows a similar situation *with* classifiers.

Example configuration does not show that for the sake of simplicity, but we could have altered it in such a way that our best customer (512 kbps) would preferentially receive any excess bandwidth unused by the other customers – if the class 1:20 would not be consuming its allocated 384 kbps and classes 1:10 and 1:30 would compete on bandwidth requirements, all unused 384 kbps would go to the class 1:10.

Listing 2.2: HTB example class hierarchy

```
tc qdisc add dev eth0 root handle 1: htb default 1:30
tc class add dev eth0 parent 1: classid 1:1 htb rate 1024kbps ceil 1024kbps
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 512kbps ceil 1024kbps
tc class add dev eth0 parent 1:1 classid 1:20 htb rate 384kbps ceil 1024kbps
tc class add dev eth0 parent 1:1 classid 1:30 htb rate 128kbps ceil 1024kbps
```

2.3.3 Classification of packets and flows

A classful qdisc needs a *classifier* to determine which class is supposed to receive a given packet. A classifier can be usually found in two places, it is either part of a filter or it is an iptables rule with `CLASSIFY` target. A filter is composed from mandatory classifier phrase (`u32` for example) and an optional policer phrase. If a classifier matches, its owner qdisc sends the packet to the class specified by the filter. Additionally, if the filter contains a policer it is applied before the qdisc takes control. A policer executes action based on whether a rate is below or above a threshold but never delays a packet.

`u32` is probably the most common classifier. It allows matching against a specified bit pattern in a packet header. For example the following filter (listing 2.4) would match a packet with Type of Service field set to low-delay. A graphical representation of the same is seen on figure 2.8 depicting IPv4 header structure. The first line just describes that a filter should be added to the root qdisc (1:0 or 1:) of interface eth0 with matching priority 1 (prio parameter specifies the order in which filters will be matched) and the filter type will be `u32`. The second line is the actual rule. The second "`u32`" specifies that 32 bit sample will be matched. `00100000 00ff0000 at 0` is the actual `u32` sample – the first number is the desired value, the second is mask and the third is offset as seen on figure 2.8. `flowid` specifies a class that will receive a matched packet. Multiple match clauses can be combined together to form more specific rule. If a higher layer match is needed (e.g. TCP) a nexthdr flag (e.g. `at nexthdr+0`) or a different sample type could be used – `u32` filter can have other samples than `u32`, such as `ip` (see listing 2.3) or `tcp`. Further description is available in LARTC [27], chapter 12. `u32` is backed by hashtables and is therefore extremely efficient. In case this would not be enough, it is possible to compile a filter as a kernel module using tcng [11].

Listing 2.3: TC filter for destination IP address

```
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match ip dst 10.0.0.10/32 flowid 1:1
```

Listing 2.4: TC filter for low-delay ToS packet

```
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match u32 00100000 00ff0000 at 0 flowid 1:2
```

Other classifiers can be used in conjunction with iptables. `fw` classifier matches against a mark set by iptables `MARK` target (see listing 2.5). iptables target `CLASSIFY` allows to directly set class for a matched packet without creating a tc filter. Main advantages of using iptables are stateful matching and far greater familiarity with creating rules among users. The problem with iptables is their reliance on Netfilter and therefore iptables matching cannot be used on IFB devices because they purposefully omit calling Netfilter hooks.

Listing 2.5: TC filter for iptables `MARK`

```
iptables -A POSTROUTING -t mangle -o eth0 -j MARK --set-mark 0x42
tc filter add dev eth0 protocol ip parent 1:0 prio 1 handle 42 fw \
   flowid 1:3
```

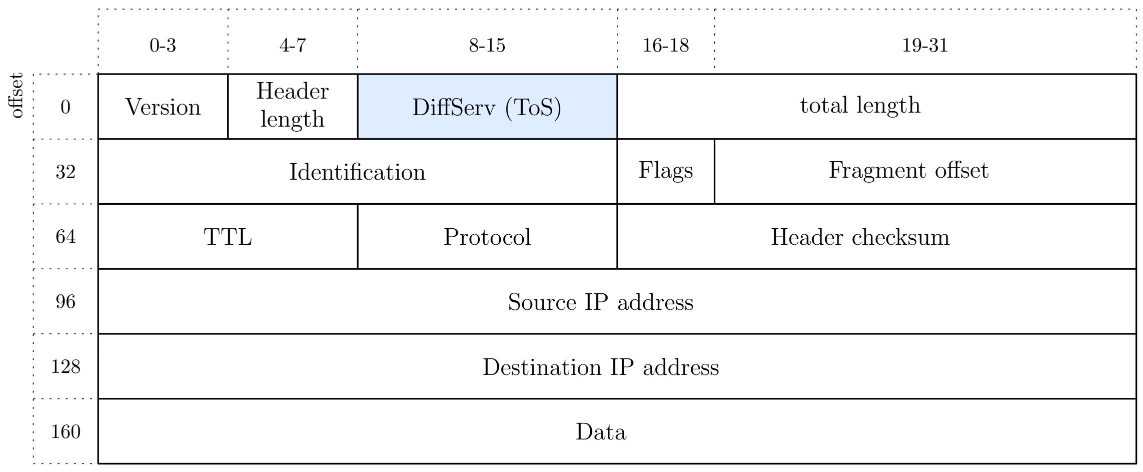

Figure 2.8: IPv4 header

2.4 Intermediate Functional Block

In terms of traffic control Intermediate Functional Block behaves very similarly to a standard network interface. You can assign queueing disciplines, classes and filters to your heart's content or sniff on it with Wireshark. There are some limitations stemming from the lack of Netfilter hooks – it is not possible to match ip_tables rules on an IFB device as the hook simply does not get called. It is deliberate decision to keep the IFB code simple[6]. However, traffic can be matched with traffic control filters and it is possible to get some ip_tables functionality through "action ipt -j [target]" as shown in IFB tutorial [24]. Behavior of the IFB can be similar to the loopback interface – if a packet is *redirected* to the IFB device, it is returned back to the original point of redirection on dequeueing.

CWC testing environment required full shaping options even for ingress traffic but standard Linux FE implementation does not support classful queueing disciplines (qdiscs) on ingress. The solution is to redirect traffic from/to the shaped nodes on ingress of all interfaces[7] to the IFB device. This automatically provides traffic aggregation and also solves the problem with ingress limitations because it is now possible to approach ingress traffic as an egress traffic thus allowing for expanded shaping features. The layout of egress on IFB in terms of TC components is depicted on figure 2.10.

An IFB interfaces (their number is controlled by a modprobe parameter) can be created by loading the *ifb* kernel module. There can be at most 16 IFB devices but just two were needed – one for download and one for aggregating upload. Apart from abovementioned resource sharing and overall increase in configuration flexibility it is possible to use IFB for logging purposes using mirror action. The IFB was added in Linux 2.6.16 (March 2006) and is therefore widely available in any modern Linux distribution.

[6]IFB is extremely lean. It comprises of roughly 200 lines of code while keeping good SMP behavior and being full-fledged replacement for the IMQ which was never incorporated into the mainline kernel because it introduced intrusive changes.

[7]Usually a loopback interface and a regular network interface because local nodes communicate through the loopback interface.

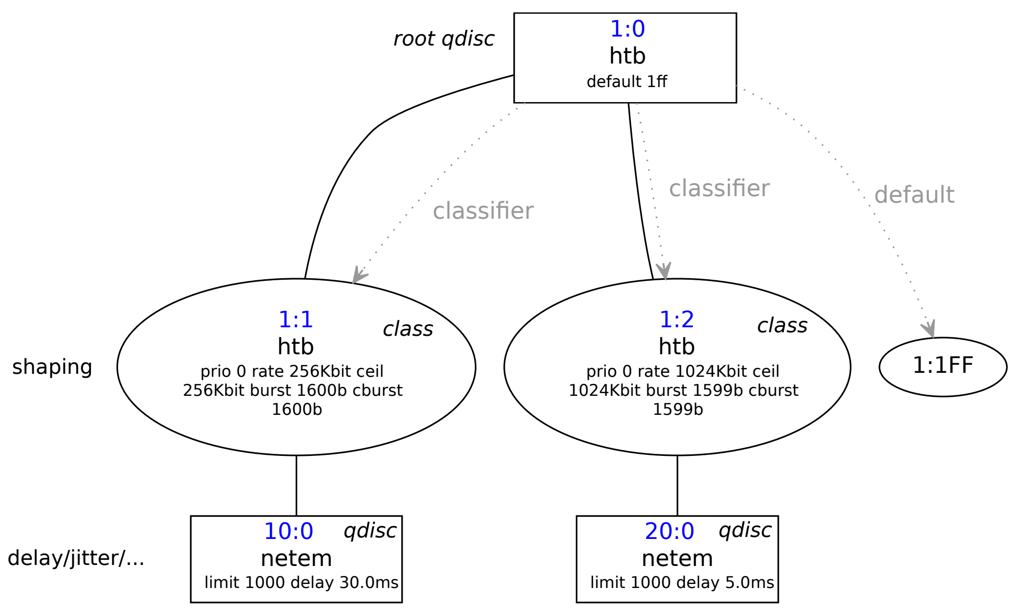

Figure 2.9: Traffic Control components inside IFB device

2.5 NetEm

NetEm [28, 29] is Linux Traffic Control queueing discipline providing features for network emulation of wide area networks. It was originally conceived for the purpose of testing Linux 2.6 TCP Vegas implementation and currently supports variable delay, loss, duplication and re-ordering of packets. Availability starting from Linux 2.6.7 (June 2004) guarantees good support in any modern GNU/Linux distribution. The simplest case is adding a fixed delay to every packet that reaches a netem qdisc. Of course this does not reflect how real networks behave but for the sake of repeatable measurements and simplicity we used just this. More advanced techniques are available for the advanced simulation – variance, correlation and Gaussian distribution. Variance and correlation allow for variable delay within the given range while the delay of the current packet is dependent on the delay of the previous one. Gaussian distribution allows even more complex patterns. Since Linux 2.6.29 NetEm is a classless qdisc [30].

A distinctive feature of NetEm is its seamless integration into the operating system allowing for network emulation even within the scope of local system between individual processes thus reducing the need for a virtualization. Commercial solutions (such as from iTrinergy [31]) take the form of separate, often rack-mounted devices and are always insanely expensive. Academic solutions such as PlanetLab or Emulab are a far better match for the testing needs of Cooperative Web Cache. However, PlanetLab [32] is relatively costly in terms of required hardware and so far we have not gained access to the Emulab. Combination of NetEm and external network emulator (possibly also NetEm) could be very powerful and make a simulation of hundreds of nodes a breeze. It should be noted NetEm has a little problem with packet loss when used locally because discarding a packet in the kernel

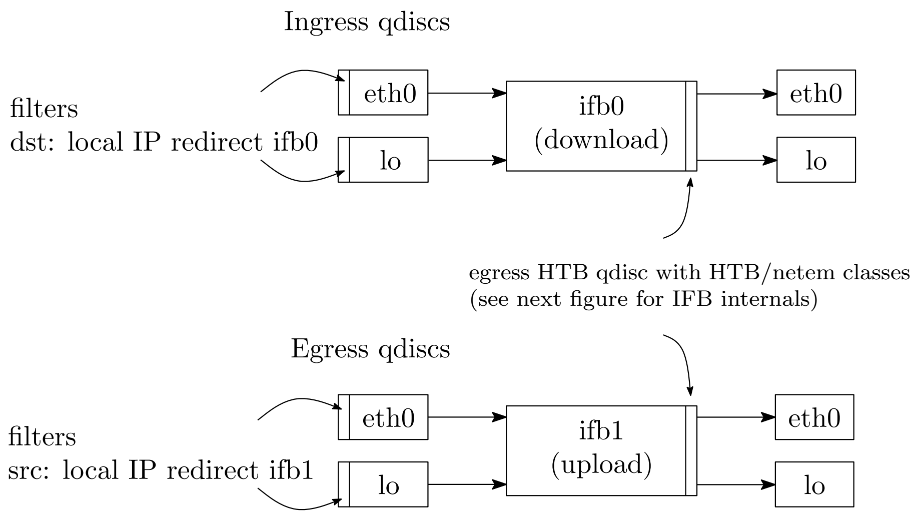
Ingress qdiscs
filters
dst: local IP redirect ifb0
eth0
lo
ifb0
(download)
eth0
lo
egress HTB qdisc with HTB/netem classes
(see next figure for IFB internals)
Egress qdiscs
filters
src: local IP redirect ifb1
eth0
lo
ifb1
(upload)
eth0
lo

Figure 2.10: Traffic Control schema

immediately notifies the upper layer that the packet was lost and this might result (for example if the upper layer is TCP) in resending and eliminating the effect of the dropping.

Chapter 3

ShaPy: Framework and Emulation

This chapter begins with two short examples of ShaPy, the first shows low-level usage of the ShaPy Framework while the second shows ShaPy Emulation – a high-level abstraction used to effortlessly create an emulated network. The chapter then proceeds to discussing settings facility, testing and verification of the ShaPy and finally some Netlink implementation details. Testing and verification section points out some important aspects of TCP behavior in relation to the network emulation.

3.1 Overview

The second goal of this project is creating a framework for programmatic control of Linux Traffic Control capabilities – ShaPy. ShaPy consists of two parts: framework and emulation. The Framework (package `shapy.framework`) comprises of a set of classes representing various elements of Linux Traffic Control (Qdisc, Class, Filter, Interface) that can be tied together in a tree structure with interface as a root. Listing 3.1 shows identical configuration as shown on figure 2.7 and listing 2.2 but written in Python using ShaPy. The second part (package `shapy.emulation`) builds on Framework and enables notation shown in listing 3.2. You can simply state what upload and download speed should be available for a transmission of packets with the given IP address. ShaPy Emulation also supports traffic aggregation from multiple interfaces as explained in 2.4. Unit tests (tests/), examples/ directory and the code itself contain useful examples.

3.2 ShaPy Framework

ShaPy Framework provides basic building blocks for programmatic creation of Linux Traffic Control schemes. ShaPy Emulation is the prime example how the framework can be used to build a more complex and user-friendly system. The framework consists of elements (classes) modelling their counterparts in the kernel – interfaces and qdiscs with filters and classes. The framework elements encapsulate operations such as addition, tearing down or statistics collection. Overview of the elements can be seen in class diagram on figure 3.1. On a theoretical level, these elements have been explained in section 2.3, this chapter is more concerned with practical implementation.

```
eth0 = Interface('eth0')
h1 = HTBQdisc('1:', default_class='1:30')
h1.add( FlowFilter('src 10.0.0.1', '1:10', prio=1) )
h1.add( FlowFilter('src 10.0.0.2', '1:20', prio=1) )

h11 = HTBClass('1:1', rate=1024, ceil=1024)
h11.add( HTBClass('1:10', rate=512, ceil=1024) )
h11.add( HTBClass('1:20', rate=384, ceil=1024) )
h11.add( HTBClass('1:30', rate=128, ceil=1024) )

h1.add( h11 )
eth0.add( h1 )
eth0.set_shaping()
```

Listing 3.1: ShaPy emulation

```
shaper_conf = {
    ('10.0.0.1', ) : {'upload': 256, 'download': 1024, 'delay': 5},
    ('10.0.0.2', ) : {'upload': 128, 'download': 512, 'delay': 30},
}
sh = Shaper()
sh.set_shaping(shaper_conf)
```

Listing 3.2: ShaPy emulation

The first element we will discuss is Interface. It represents any interface other than IFB, including loopback. Elements can be executable or non-executable meaning whether some action is needed for their initialization (executable) or they are just representation of an already existing object (non-executable). In contrast to ordinary interface, an IFB interface needs initialization – a kernel module has to be loaded and new interfaces have to be subsequently brought up. Interface elements specifically encapsulate teardown operations. In case of ordinary interfaces it means deleting ingress and root egress qdiscs, for IFB it means unloading the *ifb* kernel module (unloading kills the associated qdiscs, they are not persistent across module reloads). The Interface serves as the root of element hierarchy, it directly contains the root egress and ingress qdiscs. Upon execution of the `set_shaping()` method, the interface descends through the root elements to the underlying hierarchy and progressively executes each element. Note: the Interface class is keeping a map of already created instances with their respective names ("lo", ...) – the first call to `Interface('lo')` creates an instance but subsequent calls with the same interface name just return existing instance from the map. `Interface('eth0')` creates new instance. This is mainly for convenience (no need for meticulous reference keeping) and avoiding confusion in case the user would accidentally instantiate two eth0 interfaces.

The second element is Qdisc. As explained in section 2.3.1, a qdisc can be either classless or classful and ingress direction can contain only classless qdiscs. A classful qdisc can have filters and classes. The classes serve as a metaphorical "sockets" for other qdiscs. You cannot add one qdisc directly to the another – one or more classes have to be between them. Classes

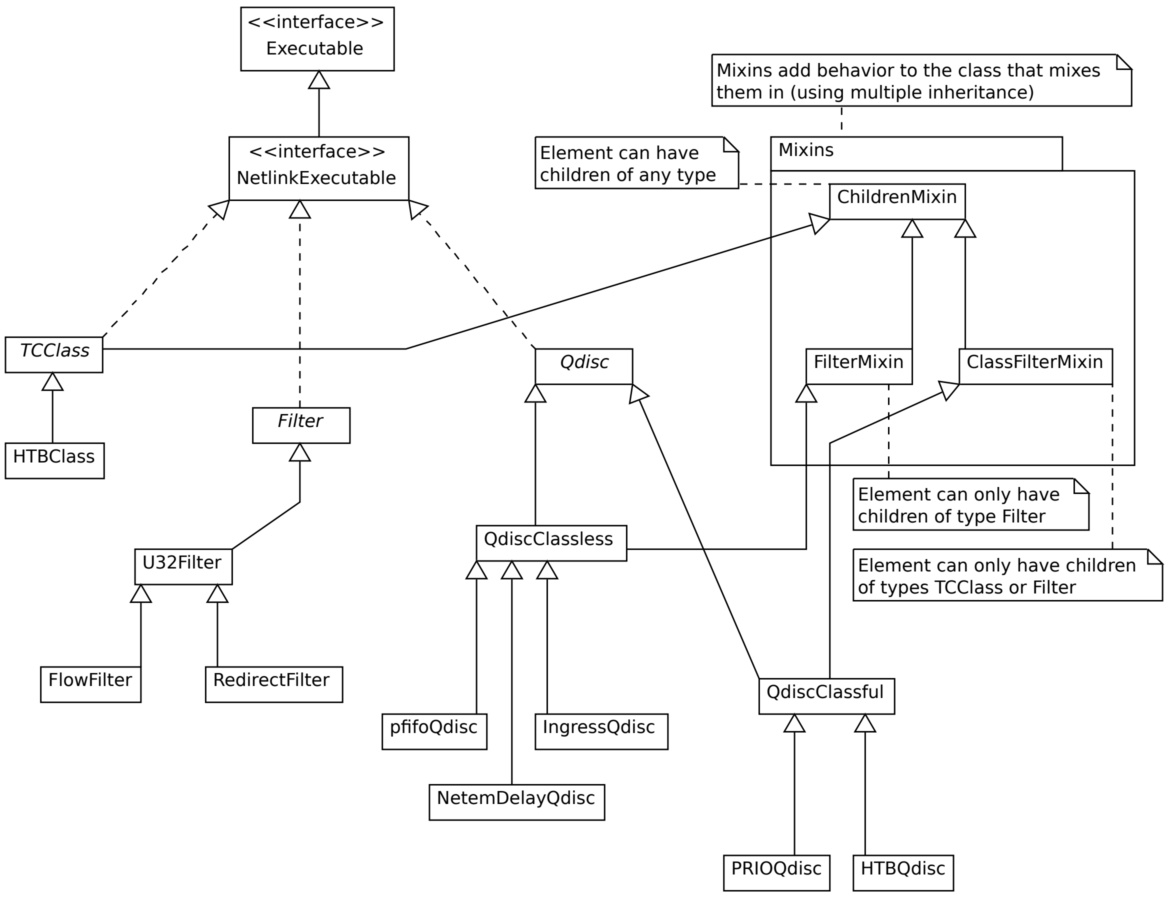
<<interface>> Executable
<<interface>> NetlinkExecutable
Mixins add behavior to the class that mixes them in (using multiple inheritance)
Element can have children of any type
Mixins
ChildrenMixin
TCClass
Filter
Qdisc
FilterMixin
ClassFilterMixin
HTBClass
Element can only have children of type Filter
U32Filter
QdiscClassless
Element can only have children of types TCClass or Filter
FlowFilter
RedirectFilter
pfifoQdisc
IngressQdisc
QdiscClassful
NetemDelayQdisc
PRIOQdisc
HTBQdisc

Figure 3.1: ShaPy overview – class diagram

are quite similar to Qdiscs from architectural standpoint with the exception that a class can contain any other element (an another class, a filter or a qdisc). Hierarchical behavior of qdiscs and classes is provided by mixin classes. A brief discussion of mixins follows.

> We can implement common methods in a class and inherit them everywhere they are needed. Such classes are commonly called *mixin* classes because their methods are "mixed in" with other classes. Mixins serve to package generally useful tools as methods. The concept is almost like importing a module, but mixin classes can access the subject instance, `self`, to utilize both per-instance state and inherited methods.
>
> – *Programming Python [33], page 599*

Mixins basically encapsulate behavior. In our case, there are three mixin classes pertaining to the hierarchical behavior. `ChildrenMixin` provides the logic of bi-directional relationship between parent and child elements. `FilterMixin` and `ClassFilterMixin` just inherit from `ChildrenMixin` and make additional checks to ensure hierarchy correctness.

ShaPy has two mechanisms to execute an *executable* element. The first is running a command associated with the element. The second is sending a Netlink message. Class `Executable` is a base class representing an object whose initialization requires running a certain command (e.g. IFB initialization requires executing command “`modprobe ifb`”). It contains logic for selecting correct command templates from the module specified by the `settings.COMMANDS` directive. See ShaPy Emulation implementation for an example how to define your own commands – in this particular case, ShaPy Emulation defines a new statistics collection command. After selecting which command template is the right one (selection is based on the element’s class name, e.g. `pfifoQdisc`), the template is filled in and executed using *subprocess*[1] module from the standard Python library. The second option for initializing an element is using a Netlink socket. Class `shapy.framework.netlink.NetlinkExecutable` represents such elements. The element itself supplies parameters – typically a message type (e.g. `RTM_NEWQDISC`) and attributes.

Using Netlink socket to create or in any way manipulate traffic control elements requires root privileges. Since ShaPy is all about manipulating traffic control elements it needs root privileges when accessing the netlink socket. Creating a connection as root and then dropping privileges does not work (see `test_privileges.py` in `tests/netlink/`) since permissions are quite logically checked for every operation. If you are not comfortable running your code with root privileges (and you should not be, but sometimes there is no sensible way out) there are two general approaches you can take. Either separate the code requiring the privileges to separate script and run it using sudo. Or at the very beginning spawn a process (see Python multiprocessing module[2]) designated for running the privileged code, drop the privileges and communicate with the privileged process using some sort of interprocess communication such as shared memory or queue. The issue of privileges is not something ShaPy could or even should solve on its own – it is a job for a program using ShaPy as a library to make sure the ShaPy commands are running with root privileges.

3.2.1 Settings

Shapy framework provides a simple facility for managing settings. User creates a regular Python file accessible as a module[3]. In his main program he subsequently registers this module as shown in listing 3.3. The module name is not important but the convention is to call it `settings.py`. The custom settings file can override or add values specified in the default ShaPy settings module `shapy.framework.settings.default`.

```python
import shapy
shapy.register_settings('settings')
```

Listing 3.3: Registering the settings module

`SUDO_PASSWORD` specifies the password that can be used to gain root privileges through the sudo program (*Executable* elements). Configuring the kernel requires root privileges, if you do not provide the password in settings it will be required while executing rules. This can

[1]http://docs.python.org/library/subprocess.html
[2]http://docs.python.org/library/multiprocessing.html
[3]http://docs.python.org/tutorial/modules.html

hamper automated test running, hence this setting. `UNITS` can be either "kbps" for kilobytes or "kbit" for kilobits (per second). This setting is used whenever you specify a speed (such as rate parameter of HTBClass). `ENV` is a dictionary of environment variables used when executing elements when running any external process. In the spirit of how settings are handled, `COMMANDS` specifies the module providing additional template commands for your custom element classes. `HTB_DEFAULT_CLASS` is HTB-specific setting specifying which class should receive unclassified traffic. It does not need to be an actual class – if you specify non-existent class the traffic not matching any filter remains untouched.

```python
SUDO_PASSWORD = 'fortytwo'
UNITS = 'kbps'
ENV = {
    'PATH': '/sbin:/bin:/usr/bin'
}

HTB_DEFAULT_CLASS = '1ff'

### ShaPy Emulation settings ###

# it is advisable to set MTU of the interfaces to something
# real, for example 1500 bytes
EMU_INTERFACES = (
    'lo',
    'eth0',
)

# ports excluded from shaping and delaying (holds for in and out)
# usually used for control ports
EMU_NOSHAPE_PORTS = (8000,)
```

Listing 3.4: Example settings.py

3.3 ShaPy Emulation

ShaPy Emulation is built using ShaPy Framework. It provides a convenient notation (see listing 3.2) for creating emulation environment. Along with the simulation environment created by Slavka Jaromerska in her Bachelor thesis [8] it was the cornerstone of testing of Cooperative Web Cache [1, 2, 7]. `Shaper` class is the heart of ShaPy Emulation, it configures the Traffic Control elements provided by ShaPy Framework based on the dictionary passed to the `set_shaping(dict)` method as seen before, allows Traffic Control teardown and facilitates statistics gathering. Teardown is done through method `teardown_all()`. Statistics are automatically gathered by the IFB qdiscs, Shaper method `get_traffic(ip)` retrieves a tuple (uploaded, downloaded) [bytes]. These counters are reset on teardown. If you omit any of the values (download, upload, delay) it will be replaced with a "dummy" value – in case of speeds a maximum throughput a HTB class allows, or 0 ms delay.

3.3.1 ShaPy Emulation settings

ShaPy Emulation also has its own settings. `EMU_INTERFACES` specifies which interfaces should be aggregated using IFB device (see section 2.4 for detailed explanation with figure). `EMU_NOSHAPE_PORTS` are exempt from shaping/emulation. This is typically used for command & control ports of instances.

3.4 Testing and verification

ShaPy contains a comprehensive set of unit tests. Listing 3.5 illustrates how Python *unittest* module can be used to discover and run unit tests for ShaPy Framework and ShaPy Emulation. A third test suite tests/netlink/ is also present, it tests some Netlink API functionality but is currently rather holey in terms of coverage.

```
sudo python -m unittest discover -c -t . -s tests/framework/
sudo python -m unittest discover -c -t . -s tests/emulation/
```

Listing 3.5: Running unittests

Iperf [34] was extensively used for manual testing. Listing 3.6 shows two clients with 256 KB/s upload and RTT 60 ms sending 2 MB of random data to the server with 256 KB/s download and RTT 10 ms. You can see the bandwidth sharing is almost perfect. This seems to be universal as all tests shown basically the same fair behavior with respect to sharing. Therefore it can be inferred with reasonable certainty the shaping does not negatively affect fairness in the network. Window sizes were intentionally kept small relative to the transferred files to keep buffering effect (described later in the chapter) from skewing upload speeds seen on the clients.

Listing 3.6: Iperf server with multiple clients

```
$ iperf -B 127.0.0.2 -s -f KB -p 8001 -w 32K
------------------------------------------------------------
Server listening on TCP port 8001
Binding to local address 127.0.0.2
TCP window size: 64.0 KByte
------------------------------------------------------------
[ 4] local 127.0.0.2 port 8001 connected with 127.0.0.3 port 8001
[ 5] local 127.0.0.2 port 8001 connected with 127.0.0.4 port 8001
[ ID] Interval Transfer Bandwidth
[ 4] 0.0-16.0 sec 2048 KBytes 128 KBytes/sec
[ 5] 0.0-16.0 sec 2048 KBytes 128 KBytes/sec


$ iperf -c 127.0.0.2 -B 127.0.0.3 -p 8001 -f KB -n 2M -w 16K
------------------------------------------------------------
Client connecting to 127.0.0.2, TCP port 8001
Binding to local address 127.0.0.3
TCP window size: 32.0 KByte
```

```
------------------------------------------------------------
[ 3] local 127.0.0.3 port 8001 connected with 127.0.0.2 port 8001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0-15.8 sec 2048 KBytes 130 KBytes/sec


$ iperf -c 127.0.0.2 -B 127.0.0.4 -p 8001 -f KB -n 2M -w 16K
------------------------------------------------------------
Client connecting to 127.0.0.2, TCP port 8001
Binding to local address 127.0.0.4
TCP window size: 32.0 KByte
------------------------------------------------------------
[ 3] local 127.0.0.4 port 8001 connected with 127.0.0.2 port 8001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0-16.0 sec 2048 KBytes 128 KBytes/sec
```

You need to be aware of bandwidth-delay product ([35], page 201) issues of TCP when determining amount of latency you will emulate between the IP addresses in your virtual network. RFC 1323 [36] states:

> TCP performance depends not upon the transfer rate itself, but rather upon the product of the transfer rate and the round-trip delay. This "$bandwidth * delay$ product" measures the amount of data that would "fill the pipe"; it is the buffer space required at sender and receiver to obtain maximum throughput on the TCP connection over the path, i.e., the amount of unacknowledged data that TCP must handle in order to keep the pipeline full. TCP performance problems arise when the $bandwidth * delay$ product is large. We refer to an Internet path operating in this region as a "long, fat pipe", and a network containing this path as an "LFN" (pronounced "elephan(t)").
>
> – *RFC 1323*

It is very easy to unknowingly create an LFN with ShaPy. Default receive/send buffer (`net.core.rmem`/`net.core.wmem`) size in Linux is 87,380/16,384 bytes (see listing 3.7). If you set the speed on your receiving side with 85.3 KB buffer to 1024 KB/s (8 Mb/s)[4] you could run into LFN problem with one-way delay of 42 ms (remember that the *delay* in the $bandwidth * delay$ formula is round-trip time). However, modern operating systems employ TCP window scaling introduced by RFC 1323. The default maximum size of scaled window on Linux is 108 KB (this value can vary depending on available memory, version and other settings, see tcp(7) for details) therefore the earlier 42 ms limit is raised to 105 ms. You should also take into consideration Nagle's algorithm (RFC 896 [37]) adds some delay, too.

It should be noted that the shaping effectiveness is partially dependent on protocol "responsiveness". For example, ShaPy would properly limit UDP packet rate but the sender would never know her packets are dropped, fails to adjust the packet rate and continues transmitting at a high speed.

[4] Actually a very common emulation setting, see the case study in [1]

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
>>> s.getsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF)
87380
>>> s.getsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF)
16384
```

Listing 3.7: Default sizes of socket buffers

Both ShaPy Framework and ShaPy Emulation have extensive unittest coverage. Throughput testing of most Framework components and large part of Emulation is done by opening two sockets. First, the *server socket* is bound in the separate thread and the test setup method prepares a buffer with random data. The contents of the buffer are then sent by the test method through its *client socket* to the server. Upon receiving the entire content, the server sends the content back to the client. It is important to realize the perceived speed on one end of the connection can be, especially for shorter transfers, quite different from the perceived speed on the other end. This is caused by buffering – if you are sending 1 MB of data and your buffer is 2 MB large your socket will report a seemingly instantaneous transfer disregarding your shaping. The shaping is of course applied but occurs when the kernel dequeues the data. In the real world this is desirable behavior but a correct bandwidth throughput measurement requires the sender to wait for the response from its peer. In our case this is done by calling recv() and waiting until the other end either closes the connection or sends our data back. Unit testing latency (delay) is done by pinging (ICMP echo) several times from a shaped to another shaped IP address. The tests can also serve as a good example how to use the framework. ShaPy uses standard Python logging facilities, the root logger for the project is *shapy*. ShaPy Emulation by default logs to file shaper.log in the current working directory. This behavior can be easily altered by swapping in different handler using standard Python logging facilities. Listing 3.8 shows an example of a simple logging configuration that will log all messages to stdout.

```
import logging
logging.basicConfig(level=logging.INFO, datefmt='%H:%M:%S',
                    format='%(asctime)s %(name)s %(levelname)s: %(message)s')
```

Listing 3.8: Setting up logging

3.5 Netlink implementation details

The ShaPy Framework heavily relies on the Netlink interface to provide majority of its functionality. It is of interest even as a general interface to the Netlink and can be used separately from the rest of the framework. This section will therefore describe some essential implementation details of the Netlink interface in ShaPy Framework. Package `shapy.framework.netlink` contains a relevant code. Figure 3.2 outlines the architecture. First, the framework needs to create a socket and connect it to the kernel. Listing 3.9 shows a relevant excerpt from the Connection class code dealing with initiating the socket. Socket buffer size is lowered

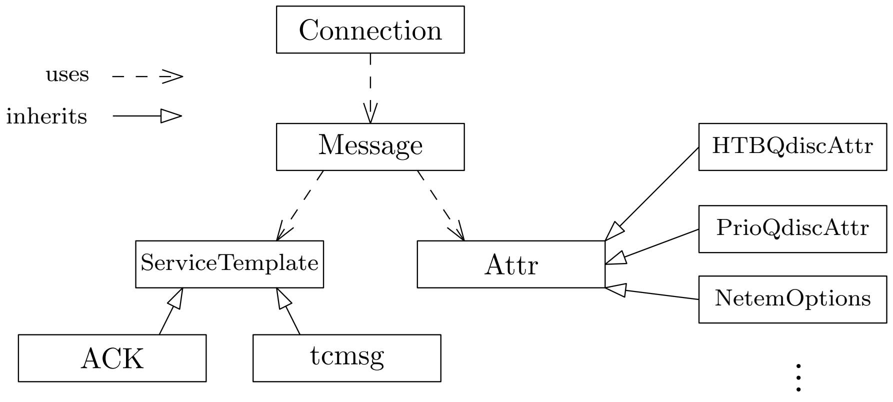

Figure 3.2: ShaPy Netlink implementation schema

to 65536 bytes since this is maximum size of the Netlink packet. Bind is called with PID 0 which lets kernel assign the correct PID to our socket. Groups parameter allows subscribing to one or more multicast groups but this functionality is not currently used in the framework. The kernel (or more precisely the FEC) does not care about sequence numbers, it just increments them when responding. However, the developer can use them to implement reliable protocol.

```
# shapy.framework.netlink.connection
class Connection(object):
    """Object representing Netlink socket connection to the kernel."""
    def __init__(self, nlserv=socket.NETLINK_ROUTE, groups=0):
        self.fd = socket.socket(socket.AF_NETLINK, socket.SOCK_RAW, nlserv)
        self.fd.setsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF, 65536)
        self.fd.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, 65536)
        self.fd.bind((0, groups))
        self.pid, self.groups = self.fd.getsockname()
```

Listing 3.9: Initializing a Netlink socket connection

Listing 3.10 shows the process of constructing a Netlink `Message` for creating a priority FIFO qdisc (`pfifo`). A Netlink message consists of three parts: a header, a service template and attributes (see figure 2.3). I will start from the middle with service template `tcmsg`. `tcmsg` accepts a set of five values, the important ones are interface index[5], handle (or classid) and parent element. Family is always unspecified (`AF_UNSPEC`) and info is always zero in the CP $\rightarrow$ FE direction. Apart from its own fields `tcmsg` class constructor accepts a list of attributes. In our case, attribute type `TCA_KIND` will be needed to specify the desired qdisc “pfifo” (asciiz string). Attribute types are dependant on the IP service message type (e.g. `RTM_NEWQDISC`), Traffic Control IP service has 7 general attribute types, see man page

[5]Interface index (ifindex) is a unique number assigned to a network interface by the kernel. This number can be found out using C function if_nametoindex() or far more easily parsed from the output of command “ip link show dev $device” – its the number before the interface name. Loopback interface *lo* has ifindex==1.

```python
from shapy.framework.netlink.constants import *
from shapy.framework.netlink.message import Message, Attr
from shapy.framework.netlink.tc import tcmsg

tcm = tcmsg(socket.AF_UNSPEC, if_index, handle, TC_H_ROOT, 0,
            [Attr(TCA_KIND, 'pfifo\0')])

msg = Message(RTM_NEWQDISC,
              flags=NLM_F_EXCL | NLM_F_CREATE | NLM_F_REQUEST | NLM_F_ACK,
              service_template=tcm)

from shapy.framework.netlink.connection import Connection
conn = Connection()
conn.send(msg)
response = conn.recv() # ACK message
```

Listing 3.10: Constructing a Message

rtnetlink(7) for their list. In addition, some components have specific attribute types (e.g. HTB class has 4). These specific attributes are usually documented only as structs in kernel header files. Finally we need to create the message itself. `Message` encapsulates the entire Netlink message. When asked to pack by the `Connection`, it packs itself and propagates request to the service template which in turn asks attributes do pack themselves. The service template then packs itself, appends packed attributes and returns control to the `Message` and ultimately the `Connection`. A Netlink message has several values (fields) of its own. The first is message type which is important for recognizing how the attributes should be interpreted. The type can be general (such as `NLMSG_ERROR`) or IP service-specific, such as `RTM_NEWTQDISC` which is specific to `NETLINK_ROUTE` IP service. The second value represents flags. The flags influence behavior of the receiver. For example, a message requesting something (e.g. creation of a qdisc) has flag `NLM_F_REQUEST` while `NLM_F_ACK` tells the receiver to acknowledge the request. The flags are just a numeric values ORed together – a standard mechanism. Some flags are also request-type-specific, for example `NLM_F_ROOT` flag specifies the "GET" request should return entire hierarchy and `NLM_F_REPLACE` specifies the "NEW" request (e.g. `RTM_NEWTQDISC`) should replace an existing object. The `tcmsg` with the assorted attributes also needs to be finally added to the `Message`.

After receiving the message FE will respond with an ACK message. ACK message is standard Netlink message of type `NLMSG_ERROR` but instead of service template and attributes it contains 4 byte error code and the Netlink message that the ACK is confirming. Please refer to RFC 3549 for more details on ACK message.

Package `shapy.framework.netlink.constants` contains all relevant constants the framework uses, gathered from various kernel header files. Most rtnetlink constants can be found in the kernel header file `include/linux/rtnetlink.h`. Associated C structures can be found in `include/linux/pkt_sched.h`. Grep can be used to effectively search through kernel header files like this: "grep -E -R -nH -i <searchterm> include/". Also, *iproute2* is the de facto standard and proved rtnetlink implementation and sometimes it is very handy to see how the

```
sudo strace -x -s 128 -e trace=send,recv,sendmsg,recvmsg -o qdisc_add.txt \
    tc qdisc add dev lo root handle 1: htb default 1ff
```

Listing 3.11: Sniffing Netlink sockets

tc or *ip* behave in certain situations. Listing 3.11 shows how `strace` can be used to capture what exactly the *tc* sends and receives from the Netlink socket in a file. See the strace man page for detailed explanation of the switches involved.

3.6 System requirements

Requirements for running ShaPy are quite slim. The basic requirement is a modern Linux kernel. Linux 2.6.29 and possibly even lower should be enough but all testing was done on Linux 2.6.35 and 2.6.38. Python 2.6 is required, 2.7 is highly recommended especially in case you want to run tests because Python 2.7 includes a new version of the unittest module. Backported module unittest2 can be installed in earlier versions but its use is not supported. Python 2.7 is stable release that has been out in the wild from July 2010 and will receive very long-term support as it is the last version in Python 2.x series. Hence requiring it as dependency is sensible. Compatibility with Python 3.x with 2to3 tool is untested but I can see no reason for Python 3 to pose any significant issues the 2to3 tool could not handle. The testing was made with iproute2 version 20091226 and 20100519. Any recent Linux distribution should easily satisfy all the requirements above. Regarding hardware architecture, the testing was done on both x86 and x86-64. Other architectures were untested due to lack of appropriate hardware and practical need.

3.7 Installation

ShaPy Framework and ShaPy Emulation are packaged together in one Python package called `shapy`. The distribution package is available through PyPI repository:
`http://pypi.python.org/pypi/ShaPy`.
You can either download, unpack and install it manually or comfortably install using pip as shown on listing 3.12. Source code repository is located on GitHub:
`https://github.com/praus/shapy`.

```
# directly through easy_install
tar xvpzf ShaPy-<version>.tar.gz && cd ShaPy-<version>
python setup.py install
# OR from PyPI
pip install ShaPy
# OR directly from project's GitHub repository into your virtualenv?
pip install -e git://github.com/praus/shapy.git#egg=shapy
```

Listing 3.12: ShaPy installation

Chapter 4

Conclusion

Motivation for this work originated in a practical need to emulate a complex peer-to-peer network called Cooperative Web Cache. Simulation with meaningful fidelity would be too impractical due to complexity of the researched peer-to-peer network. Therefore our team decided to evaluate the CWC using emulation instead. It turned out there were no viable emulation frameworks allowing network separation in context of one operating system instance. Our hardware resources were quite limited and since we absolutely needed to run multiple nodes[1] on one machine we set on building our own solution that would allow us to run at least a limited emulation environment entirely locally. This thesis described low-level parts[2] (collectively called ShaPy) of the solution developed for testing Cooperative Web Cache. The ShaPy is basically concerned with configuring the Linux kernel through the Netlink interface with the ultimate goal of creating a local virtual network. Chapter 2 extensively discussed theoretical background necessary for understanding the concepts behind the used components. Chapter 3 explored implementation details of ShaPy while pointing out some caveats and common pitfalls. Overall, the goals set at the beginning in section 1.2 were met. The ShaPy is general framework offering a reasonably realistic emulation under wide range of emulation scenarios while being easily extensible to accomodate new requirements.

4.1 Future work

At present, ShaPy is only capable of emulating homogenous virtual network in terms of link latency. This means each node has fixed delay in all directions and thus it is not possible to create clusters of nodes that are close to each other but distant to other nodes outside the group. The required functionality is there but was not implemented so far. Most functionality in ShaPy is built around generic, object-oriented Netlink interface which could be leveraged for more than just Linux Traffic Control. Unit test coverage could use some improvement, especially for Netlink interface code and some more complex interactions

[1]Or more generally an application instance identified by an IP address.

[2]High-level parts were written by Slavka Jaromerska. Her framework P2PEM is concerned with controlling multiple networked machines with the ultimate goal of creating one unified emulation environment. Let's take an example: nodes A and B are running on pc1 and node C is running on pc2. P2PEM would use the ShaPy on each machine to create a virtual network where node A would experience very similar network properties when communicating with node C as opposed to the "local" node B.

between components. Also, the current implementation of the classes modelling the traffic control elements does not support all features of the underlying elements in the kernel.

Finally, the aspect of emulation fidelity was not directly addressed in this thesis because it is slightly out of its scope. However, HTB and Netem (the main components the ShaPy relies upon for its functionality) are standard components that were present in the kernel for many years which guarantees a certain level of software maturity and resulting reliability. Their characteristics are relatively well known and can be easily found in the respective documentation. While using the ShaPy for testing of Cooperative Web Cache I did not ran into any problems related to these components. Although the absence of thorough case study does not limit practical usefulness of the ShaPy for the reasons explained above, it could be interesting topic for future work.

Bibliography

[1] Tomas Cerny, Petr Praus, Slavka Jaromerska, Lubos Matl, and Jeff Donahoo. Cooperative web cache. In *18th International Conference on Systems, Signals and Image Processing*, 2011.

[2] Tomas Cerny, Slavka Jaromerska, Petr Praus, Lubos Matl, and Jeff Donahoo. Cooperative web cache. submitted for peer-review on P2P11 conference, 2011.

[3] dummynet. `http://info.iet.unipi.it/~luigi/dummynet/`.

[4] W. Almesberger et al. Linux network traffic control—implementation overview. In *Sixth IEEE Symposium on*, pages 296–301, 2001.

[5] Jamal Hadi Salim. IP bandwidth management. *Linux Journal*, (Issue #62), June 1999.

[6] J. Calcote. *Autotools: A Practioner's Guide to GNU Autoconf, Automake, and Libtool*, pages 147–148. No Starch Press Series. No Starch Press, 2010.

[7] Luboš Mátl. Systém pro distribuci statických zdrojů urychlující načítání webových aplikací. Czech Technical University in Prague, Faculty of Electrical Engineering, Bachelor thesis, 2011.

[8] Slavka Jaromerska. Environment for peer-to-peer application simulation with application on cooperative web cache. Czech Technical University in Prague, Faculty of Electrical Engineering, Bachelor thesis, 2011.

[9] Dan Siemon. Linux QoS library. `http://www.coverfire.com/lql/`.

[10] Thomas Graf. libnl. `http://www.infradead.org/~tgr/libnl/`.

[11] Werner Almesberger. Traffic control - next generation. `http://linux-ip.net/gl/tcng/`.

[12] C. Kiddle, B. Unger, and R. Simmonds. Advances in Network Emulation. *Annual Review of Network Management and Security*, page 57, 2006.

[13] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989. Updated by RFCs 1349, 4379, 5884, 6093.

[14] M. Carson and D. Santay. NIST Net-A Linux-based network emulation tool. *Computer Communication Review*, 33(3):111–126, 2003.

[15] H.K. Kalita and M.K. Nambiar. Designing WANem: A Wide Area Network Emulator tool. In *Communication Systems and Networks (COMSNETS), 2011*, 2011.

[16] J.S. Ahn, P.B. Danzig, Z. Liu, and L. Yan. Evaluation of TCP Vegas: emulation and experiment. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 185–195. ACM, 1995.

[17] H.J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The microgrid: a scientific tool for modeling computational grids. 2000.

[18] S. Keshav. REAL 5.0. http://www.cs.cornell.edu/skeshav/real/overview.html.

[19] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol. RFC 3549 (Informational), July 2003.

[20] Forwarding and control element separation (forces). http://datatracker.ietf.org/wg/forces/charter/.

[21] H. Khosravi and T. Anderson. Requirements for Separation of IP Control and Forwarding. RFC 3654 (Informational), November 2003.

[22] F. Baker. Requirements for IP Version 4 Routers. RFC 1812 (Proposed Standard), June 1995. Updated by RFC 2644.

[23] Netfilter. http://netfilter.org/.

[24] Intermediate functional block. http://www.linuxfoundation.org/collaborate/workgroups/networking/ifb.

[25] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking (TON)*, 3(4):365–386, 1995.

[26] S. Misra, S.C. Misra, and I. Woungang. *Selected Topics in Communication Networks and Distributed Systems*, pages 384–385. World Scientific, 2009.

[27] Bert Hubert. Linux advanced routing & traffic control howto. http://lartc.org/howto/.

[28] Netem. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[29] S. Hemminger et al. Network emulation with netem. In *Linux Conf Au*, 2005.

[30] Netem classless commit. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=02201464119334690fe209849843881b8e9cfa9f.

[31] Network emulators from itrinegy. http://www.itrinegy.com/network-emulators/network-emulator-overview.html.

[32] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[33] M. Lutz. *Programming Python*. O'Reilly Series. O'Reilly Media, 2011.

[34] Iperf. http://iperf.sourceforge.net/.

[35] D. Medhi and K. Ramasamy. *Network routing: algorithms, protocols, and architectures*. The Morgan Kaufmann series in networking. Elsevier/Morgan Kaufmann, 2007.

[36] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992.

[37] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984.

Appendix A

Index of Abbreviations

CWC	Cooperative Web Cache
EFF	Electronic Frontier Foundation
LQL	Linux QoS Library
TCNG	Traffic Control Next Generation
RFC	Request For Comment
NIST	National Institute of Standards and Technology
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ARP	Address Resolution Protocol
FIFO	First-in First-out
HTB	Hierarchical Token Buffer
TBF	Token Buffer Filter
CBQ	Class-Based Queueing
IFB	Intermediate Functional Block
SMP	Symmetric Multiprocessing
LARTC	Linux Advanced Routing & Traffic Control HOWTO
ForCES	Forwarding and Control Element Separation, see `http://datatracker.ietf.org/wg/forces/charter/`
TLV	Type-Length Value
CP	Control Plane

CPC Control Plane Component

FE Forwarding Engine

FEC Forwarding Engine Component

NE Network Element

RTT Round-Trip Time

LFN Long Fat Network (“elephan”)

Appendix B

CD Contents

Listing B.1: CD contents

```
.
|-- examples                 example usage of the ShaPy
|-- README                   basic installation guide with examples
|-- setup.py                 distutils installation script
|-- shapy
|   |-- emulation            ShaPy Emulation
|   |   `-- commands         emulation-specific commands
|   |
|   `-- framework            ShaPy Framework
|       |-- commands         element commands for Executable elements
|       |-- netlink          Netlink interface
|       `-- settings         default settings
|
|-- text                     thesis text
`-- tests                    unit tests
```