BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

TOWARDS MORE EFFECTIVE FLOW ALLOCATION IN RINA

ZEFEKTIVNĚNÍ ALOKACE TOKŮ V RINA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MICHAL KOUTENSKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. VLADIMÍR VESELÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Koutenský Michal, Bc.**
Program: Informační technologie Obor: Počítačové sítě a komunikace
Název: **Zefektivnění alokace toků v RINA**
Towards More Effective Flow Allocation in RINA
Kategorie: Počítačové sítě
Zadání:

1. Analyzujte stávající TCP/IP řešení zabývající se sdílením šířky pásma (TCP BBR, QUIC, aj.) mezi toky.
2. Seznamte s rekurzivní síťovou architekturou (RINA) a detailně prostudujte její referenční model. Poučte se o algoritmu Raft.
3. Podle doporučení vedoucího navrhněte úpravu stávající implementace rlite o podporu alokace toků s ohledem na dostupnou šířku pásma
4. Implementujte rozšíření z bodu 3.
5. Otestujte výsledek, pokuste se o jeho srovnání s tradičními TCP/IP řešeními na prototypové topologii. Diskutujte výstupy práce.

Literatura:

- DAY, John. *Patterns in network architecture: a return to fundamentals*. Pearson Education, 2007.
- ONGARO, Diego; OUSTERHOUT, John K. In search of an understandable consensus algorithm. In: *USENIX Annual Technical Conference*. 2014. p. 305-319.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Veselý Vladimír, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 30. října 2018

Abstract

This master's thesis focuses on design and implementation of a flow allocator policy which supports bandwidth reservation for the Recursive InterNetwork Architecture (RINA). Each flow has some dedicated bandwidth, which is guaranteed to be available during the whole lifetime of the flow. The allocator, which operates as a distributed system, attempts to find a suitable path in the network graph. To achieve this goal, it must keep the information about link utilization up to date. The proposed allocator has been implemented in the open source project rlite. The first half of the thesis is concerned with congestion control theory, and also studies a number of algorithms used in TCP. Additionally, it contains an overview of the structure of RINA and the Raft consensus algorithm.

Abstrakt

Táto diplomová práca sa venuje návrhu a implementácii stratégie alokovania tokov s podporou pre rezerváciu šírky pásma v rekurzívnej sieťovej architektúre (RINA). Každý tok má vyhradenú šírku pásma, ktorej dostupnosť je garantovaná počas celej doby života toku. Alokátor, ktorý funguje ako distribuovaný systém, má za úlohu nájsť vhodnú cestu v sieťovom grafe a udržiavať informácie o využití spojov aktuálne. Navrhnutý alokátor bol implementovaný do open source projektu rlite. V prvej polovici sa práca zaoberá teóriou riadenia zahltenia a štúdiom algoritmov použitých v TCP. Práca taktiež obsahuje popis hlavnej štruktúry architektúry RINA a konsenzus algoritmu Raft.

Keywords

TCP, congestion control, RINA, flow allocation, guaranteed bandwidth, rlite

Kľúčové slová

TCP, riadenie zahltenia, RINA, alokácia tokov, garantovaná šírka pásma, rlite

Reference

KOUTENSKÝ, Michal. *Towards More Effective Flow Allocation in RINA*. Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Veselý, Ph.D.

Rozšírený abstrakt

Cieľom tejto diplomovej práce je načrtnúť nový smer efektívneho využitia šírky pásma v sieťach umožnený vďaka rekurzívnej sieťovej architektúre (RINA). Najprv sa pozrieme na riadene zahltenia z teoretického pohľadu a študujeme prečo a ako zahltenie v sieťach vzniká. Skúmame možné prístupy k riadeniu zahltenia a taktiež vlastnosti, ktoré by malo funkčné riešenie spĺňovať.

Nasleduje prehlad a porovnanie aktuálnych algoritmov pre riadenie zahltenia používaných v TCP/IP. Ako prvý je uvedený TCP Tahoe, primárne z historických dôvodov, ktorý slúži ako základný algoritmus pre riadenie zahltenia pri popisovaní toho, ako fungujú novšie algoritmy a v čom sa od Tahoe líšia. Ako študované algoritmy boli vybrané TCP CUBIC, QUIC a TCP BBR. U každého z týchto algoritmov je popísaný jeho princíp a hlavná myšlienka. Ďalej je takiež analyzovaná jeho funkčnosť (efektívne využitie šírky pásma) ako aj chovanie pri zdielaní linky s tokmi využívajúcich iné algoritmy.

Referenčný model RINA je predstavený, popisujúci základné stavebné prvky tejto architektúry a to, ako v nej prebieha komunikácia. Zvlášť pozornosť je venovaná práve tomu, ako dochádza k zahájeniu komunikácie a alokácii tokov.

V krátkosti je predstavený problém konsenzu v distribuovanom systéme, a to, ako tento problém rieši algoritmus Raft.

Jadrom práce je návrh a implementácia nového spôsobu alokácie tokov ktorá umožňuje tokom rezervovať si časť šírky pásma. Dostupnosť tejto rezervácie je garantovaná po celú dobu života toku. Keďže alokátor odmietne toky pre ktoré požadovaná šírka pásma nie je dostupná, a toky sú obmedzené len na šírku pásma ktorú si rezervovali, zahltenie by nemalo nastať. Na dosiahnutie tohoto cieľa si alokátor udržuje aktuálne informácie o využití pásma liniek, a pri požiadavku na alokáciu toku prehľadáva graf siete za účelom nájsť vhodnú cestu. Popis implementačnej časti je rozdelený na jendotlivé podproblémy, kde u každého je popísaná motivácia ako aj implementované riešenie.

Posledná kapitola obsahuje zhodnotenie dosiahnutých výsledkov a toho ako sa systém správa. Implementované riešenie umožňuje predchádzať zahlteniu ako aj efektívnejšie využívať dostupné cesty vďaka tomu že viaceré toky z toho istého zdroja k tomu istému cieľu sú schopné putovať inými cestami v sieti.

Towards More Effective Flow Allocation in RINA

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Vladimír Veselý, Ph.D. The supplementary information was provided by Vincenzo Maffione. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Michal Koutenský
May 22, 2019

Acknowledgements

Veľká vďaka patrí v prvom rade mojej rodine za podporu pri písaní tejto práce, špeciálne kedže táto práca a choroba idú ruka v ruke. Hlavná vďaka náleží Nke za všetko to skvelé jedlo ktoré som mal k dispozícii, čo mi umožnilo sústrediť sa na písane. Nedá mi nespomenúť taktiež vaňu, ktorá bola počas písania tradičným miestom odpočinku ako aj miestom kde sa často podarilo prísť na koreň aj tým najmysetrióznejším bugom.

Vďaka patrí aj vedúcemu práce, Vladimírovi, nie len za feedback a smerovanie práce, ale aj za to že bez neho by som sa pravdepodobne k RINe nikdy nedostal. Takisto vďaka nemu som mal možnosť byt dočasnou (aj keď neoficíalnou) súčasťou projektu OCARINA počas môjho pobytu v Osle, a nahliadnuť tak bližšie ak akademickému životu.

Therefore, thanks belong to the whole OCARINA team as well, who were extremely welcoming during my stay in Oslo and from who I gained lots of interesting experience. Especially Marcel (who I hope will forgive me for being included in the part written in English), with whom I had many talks about RINA (and not just RINA), and Kristian, as in their office I spent a great part of my time at UiO. Additionally, it was from them I was able to borrow a physical copy of Patterns, which made reading it even more enjoyable. Last but not least from the UiO team, my thanks go to Michael (although we met very briefly in person) for writing a wonderful book on congestion control, which forms the backbone of the theoretical part of this thesis.

On the technical side of things, a lot of gratitude belongs to Vincenzo, for writing rlite in the first place, as it is with that project I've been involved for over two years now, and hope to be for still some time. During those two years, he has helped and guided me to understand RINA from a more practical, implementational point of view, as I sometimes tend to get carried away and lost in the world of ideas. Especially during my work on this thesis, he has engaged with me in countless discussions about the flow allocator, spanning many pages of countless e-mail threads. He has continually kept guiding me back on the track of what we're actually trying to do, even though I did my best to get sidetracked repeatedly. He has also helped me greatly with the implementation, by handling some kernel parts, as well as helping me troubleshoot many of the mysterious bugs that appeared, as debugging a kernel network stack and a distributed IPC facility isn't the easiest.

As a final entry in this section, I'd like to thank my two great friends, Nic and Iga. Nic for being a shining sun of positivity under all circumstances, and for her wonderful *snek's cold kicker tea* recipe, which greatly alleviated my suffering during the Christmas holidays. Iga for the constant well-needed encouragement to keep working on things, even if "the bug makes zero sense and I have absolutely no idea why it's happening", and for generally being an absolute sweetie to be around.

Now follow two recipes, one in English, for the aforementioned tea (which I hope will be as useful to someone as it was to me), and one in Slovak for one of my favourite soups, strúčkovú polievku.

SNEK'S COLD KICKER TEA

- 2 lemons
- 1 orange
- a pinch of salt
- a pinch of cinnamon
- a pinch of cayenne pepper
- a tablespoon of honey

Cut fruit in slices, put everything in a small pot, cover with water & simmer. Add honey after 15 minutess.

STRÚČKOVÁ POLIEVKA

Zoberieme zavárané strúčky, aj s nálevom, a spolu s nakrájanými zemiakmi a udeným mäsom dáme variť. Pridáme trochu múky na zahustenie (cca 1 kávovú lyžičku). Dochutíme soľou, octom a vňaťkou.

Contents

Chapter 1

Introduction

The rise of computer networking in the 1970's has brought us a before-unseen ability to communicate all across the globe, and by now has touched almost every aspect of our lives. Chief among the achievements surely stands the Internet, a global network with a misleading name. Originally intended for scientific collaboration, the Internet now serves both as the crucial backbone of various systems that are needed for our day-to-day life and as an entertainment medium with a nearly endless stockpile of content. Ranging from international trade and business, sharing of vast amounts of data between scientists, and various sensor networks to simply streaming a movie to your phone or talking to friends half a world away, the usage of Internet has spread so wide within our society as to become ubiquitous.

The Internet has seen an unexpected growth since its inception and continues to expand at a seemingly exponential rate as more and more devices are connected every day. This success was however not without issues, and the Internet community is constantly identifying new problems and working on finding solutions that would be applicable.

The causes of this are manifold — firstly, the operating environment keeps changing. The computing power available, transmission speeds, use-cases and demands as well as the heterogeneity of its components is vastly different to how things looked when the Internet was still just a research project. It is therefore of no surprise that some mechanisms do not work well in environments they were never meant to be deployed in and might need changes, ranging from minor adjustments to complete reworks and replacements.

Secondly, at its time of creation, the ARPANET was a proof of concept. No one has ever built a switched datagram network before — in a very real sense, it is (was!) a *First System*. Very little was known about the principles of networking, so inspiration was taken for systems design and other related disciplines when possible, while the rest was left to intuition. After several decades and an immense amount of research and practical usage, it turns out some assumptions made were right, while other concepts are in fact counter-intuitive.

The third cause is closely tied to the second — the Internet is a victim of its own success. Various issues that have been known since the beginning, like the problem with multihomed addressing, were left to be solved at a later date.[6] As it is not uncommon in software engineering, that never happened, and de facto temporary solutions have been here with us for over thirty years. The sheer scale of the Internet, coupled with how it is now considered a crucial part of the infrastructure that almost everything else seems to depend on in one way or another, making radical changes extremely difficult. Researchers are limited to do mostly small incremental adjustments which work around the problem while being

backward compatible with actors that do not know about this change. Back in 1983, the network was still small enough that they could afford to do a flag day, namely, switch from NCP to the TCP/IP protocol suite across all nodes in the network. With the number of nodes in the network being currently in the billions, one can barely try to imagine the logistics of a such day-to-day switch, and the consequences of such global downtime.

I am borrowing the categorization of *First*, *Second*, and *Third System* from the wonderful *Linux and Unix Philosophy* [8], in which the author dedicates a whole chapter to this concept. To strike a balance between usefulness and brevity, I am going to cite the main traits of each system, and how they relate to computer networking.

The First system

- Man builds the First System with his back against the wall.
- He has no time to do it right.
- Man builds the First System alone or, at most, with a small group of people.
- The First System is a „lean, mean, computing machine“.
- The First System displays a concept that ignites others' imaginations.

As stated before, the original ARPANET fits this categorization. It was a result of a constrained environment with a clear and pragmatic goal, and was the basis for the vast amount of work in this field.

The Second System

- „Experts“ build the Second System using ideas proven by the First System.
- The Second System is designed by a committee.
- The Second System is fat and slow.
- The world hails the Second System as a great achievement, with much pomp and circumstance.

We can view the current, TCP/IP Internet, as an instance of this. The complexity of the current stack of network technologies means that even the theoretical five-layer model does not hold true anymore and the layer abstraction often gets broken for the sake of pragmatism.

The Third System

- The Third System is built by people who have been burned by the Second System.
- The Third System usually involves a name change from the Second System.
- The original concept is still intact and is regarded as obvious.
- The Third System combines the best characteristics of the First and Second Systems.
- Finally, the Third System's designers are usually given the time to do it right.

A *Third System* in this case would be the Recursive InterNetwork Architecture (RINA). Taking the concepts from the *First System* and lessons learned *Second System*, it tries to synthesize these into an architecture that is both full-featured and elegant. This thesis is mostly concerned with RINA, and a whole chapter is dedicated to describing it in much more detail.

No matter the actual technology, a large-scale communications network has to contend with the issues of congestion and fairness. It is generally regarded as sensible that one user should not be able to take all the resources for themselves[1], and that the network should be resilient to increases in incoming traffic, if it wants to be usable.

In TCP/IP, this is realized by *congestion avoidance* algorithms. Although there has been a great deal of research on this topic, and many different algorithms exist with various variables that can be tweaked, as we will see in the following chapter, they all work roughly in the same way: a network flow increases its transmission speed until it guesses, using some measured metric, that the path has become congested and backs off. These two phases repeat, resulting in overfilling the network and then slowing down to let it drain.

RINA allows us to try a different approach. Before allocating a flow, we can look if we can reserve some minimum guaranteed bandwidth on the path to the destination, and simply reject the request if we are unable to provide this guarantee. In this way, we can make sure that each active flow will have enough bandwidth to transmit some useful information, and that no extra flows can be created that would result in network congestion. The TCP congestion avoidance is a reactive an approach that tries to see how much it can send before having to slow down to avoid congestion, while this is a proactive approach that tries to prevent the possibility of congestion occurring in the first place. However, such approach lacks the flexibility expected of modern networks; therefore, some consideration will be given to whether these two modes could be used in collaboration.

This introductory chapter serves as an entry point that presents the context and motivation behind this work, as well as the structure of the work as a whole. It is followed by the second chapter, which studies congestion control from a theoretical point of view, looking at why and how congestion in networks happens and what approaches can be taken when trying to avoid it. The third chapter analyzes concrete congestion control algorithms that have been deployed in the Internet. Its purpose is to present an overview of current approaches to the reader. The fourth chapter serves as a brief introduction to RINA and familiarizes the reader with the overall architecture and its components, followed by a chapter describing the working of the Raft distributed consensus algorithm and the problem it solves. The sixth chapter, which is the core of this work, contains discussion about the implemented extensions to the rlite RINA implementation. The designed solution is presented by decomposing it into several sub-problems. The motivation behind solving each sub-problem is described, as is how it fits into the goal of the thesis as a whole. The chapter ends with showcasing an example operation of the implemented solution, to demonstrate how it works. The final chapter examines the achieved results, with some comparisons to existing systems. A significant part is dedicated to discussing possible enhancements and the pitfalls they contain. The thesis ends with a conclusion which summarizes the achieved results.

[1] Assuming the case that no user is privileged over another.

Chapter 2

Congestion in Computer Networks

Congestion is not a phenomenon limited to just computer networks, but one that can be observed in a wide variety of environments. Whether it includes traffic jams or overcrowded spaces at a festival, there are some common reasons why congestion happens — mainly that resource demands exceed the capacity. In this chapter, we will closely look at congestion in computer networks from a general point of view — specific TCP congestion control algorithms are discussed in a separate chapter.

2.1 The Causes of Congestion

The flow and rate of traffic in computer networks naturally fluctuate for a multitude of reasons. Some changes are regular and predictable — there are usually more users uploading and downloading data during the day than during the night, while others may be caused by quick, unpredictable events, e.g. a user trying to download a large media file from a remote storage. In the first scenario, the traffic increase lasts for several hours and therefore it makes sense to have links with high enough capacity to meet these demands. It is the second case that congestion control is more interested in — the network sees a sudden, relatively short-lived spike in traffic that saturates the link. The excess traffic is unable to be transmitted and the device is left with only two options — buffer the packets to be transferred at a later date, or drop them.

Most network-capable devices today implement some sort of queue for storing incoming packets. Thus, the practice is to buffer the packets, and only drop them if the queue is full.[34] The assumption this operates on is that once the burst increase passes, the buffered packets can be used to efficiently utilize the link capacity, which would nicely compensate for short-lived spikes. It might intuitively seem reasonable to have long queues, as it increases the chance of successfully being able to buffer and accommodate the burst traffic. This is not true for two reasons:

1. Operations on the queue add significant delay, dependent on the length of the queue.
2. Internet traffic does not strictly follow a Poisson distribution — the number of sudden increases and decreases in traffic does not have to be the same.

From these two statements, we can see that unless we have an infinitely long queue, packet loss can still occur. Furthermore, because the long delay is undesirable, queues should generally be kept short.[34]

Network links usually operate some threshold below their full capacity to allow some headroom for such traffic spikes. This is called overprovisioning and brings some benefits — besides handling small traffic bursts without saturating the link, an overprovisioned network is easier to control and provides some room to accommodate a growing userbase without having to immediately upgrade the network or failing to provide an acceptable service.

The goal of congestion control mechanisms is to utilize the network efficiently — that is, transfer as much traffic as possible while avoiding packet loss and trying to keep the delay low.[34] As we have previously mentioned, congestion leads to queues being filled, and that leads to dropped packets and increased delay. Congestion is therefore undesirable and must be avoided. The book *Network Congestion Control*[34] defines the role of congestion control as such:

> Congestion control is about using the network as efficiently as possible. These days, networks are often overprovisioned, and the underlying question has shifted from 'how to eliminate congestion' to 'how to efficiently use all the available capacity'. Efficiently using the network means answering both these questions at the same time; this is what good congestion control mechanisms do.

To illustrate how congestion arises, it provides the following example case:

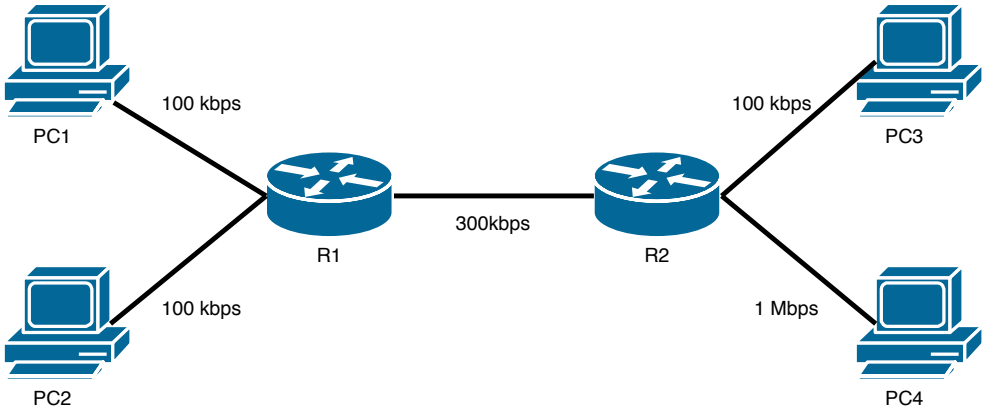

Figure 2.1: A basic network scenario

Consider that there are two flows active in the network, one from `PC1` to `PC3` and another from `PC2` to `PC4`, both operating at their full available capacity, and no congestion control mechanisms are in place. Each flow is limited to 100 kbps by the first link along the path; however, since the shared link between `R1` and `R2` is of satisfactory high capacity, no congestion happens, and each flow can utilize its path fully.

Let's say we upgrade the link between `PC1` and `R1` to 1 Mbps, resulting in the scenario found in figure 2.2. What happens is that traffic arrives at `R1` faster than it can leave and the queue starts filling. The interesting part of this experiment is that after a while, the `PC2-PC4` flow becomes starved, even though the `PC1-PC3` can also only receive 100 kbps at best. Increasing the speed of a link did not improve the overall performance of the network, quite the contrary. A plot displaying the changes of throughput over time is found in figure 2.3, generated using scripts from the book.

We can notice three distinct phases in the plot — first, both flows are continually increasing their throughput until they reach the *knee*, after which they are both transmitting

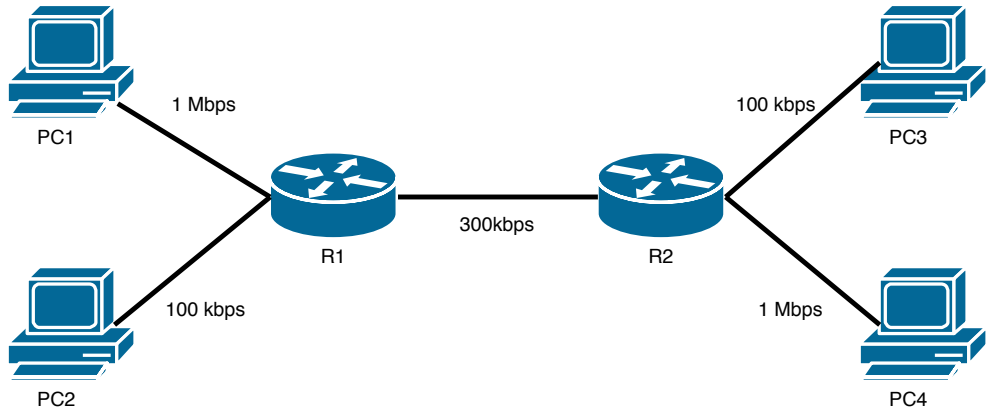

Figure 2.2: A basic network scenario with upgraded link capacity

at their maximum capable speed and the links are saturated; however, a queue at `R1` proceeds to build up because it cannot dispatch packets as fast as it is receiving them. As there is ten times more data arriving from `PC1` than from `PC2`, this ratio is represented in the contents of the queue as well. As `R2` starts to send off packets from the queue, most of them will belong to `PC1` and `PC2` will, therefore, get noticeably lower throughput. The breaking point for this is denoted as the *cliff* in the plot. It is worth pointing out that the plot shows the throughput as seen by the receivers — it is quite possible (and likely) that `PC1` will be able to achieve more than 100 kbps on the `R1-R2` link, which will result in some packets being buffered on `R2` as well. That bandwidth could be better utilized by allowing the `PC2-PC4` flow through.

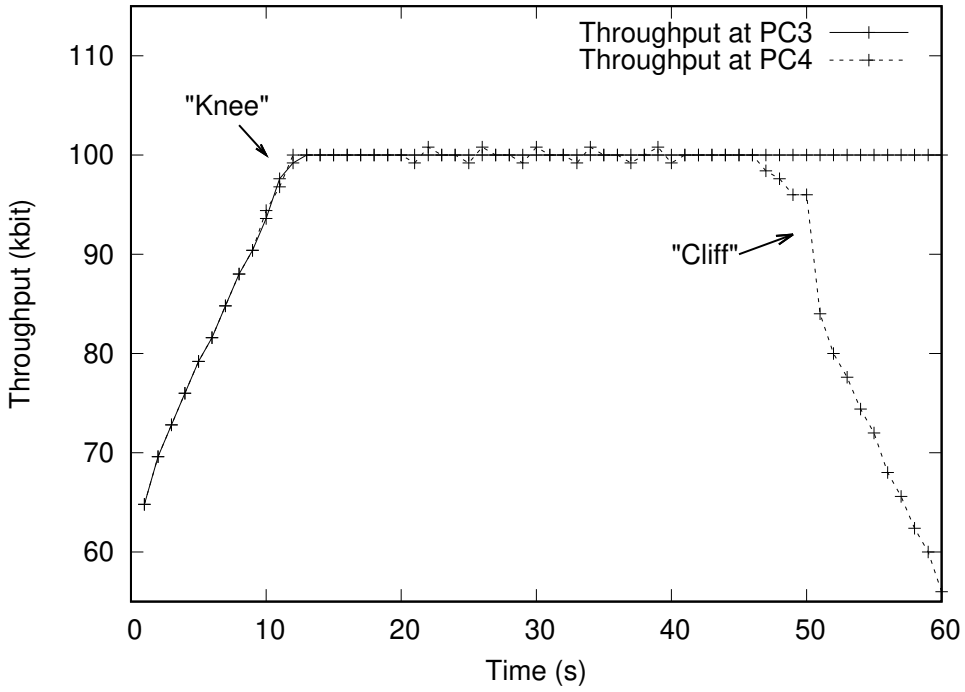

Figure 2.3: Link throughput from figure 2.2

The aim of congestion control is to have the network operate left of the cliff (for to the right lie dragons and congestion). Avoiding congestion is one part of the problem, the other part is to be efficient. The significant point here is the knee — although the overall throughput is similar from there all the way to the cliff, the further we get from the knee, the more the queues get filled, resulting in increasing delay. Therefore, most congestion control algorithms try to estimate the knee and have the senders operate near it.

2.2 Controlling Congestion

The basic idea of congestion of control is as follows: have the sender adjust its sending rate based on information about the network so that it operates near the knee. *How do we get the information*, and *how do we act on it* are the two main problems every congestion control algorithm tries to solve.

There exist two main approaches to controlling congestion.[17] The first is *proactive*, where each user gets reserved some amount of bandwidth that is guaranteed to be used solely by them. This is how things were traditionally done in circuit-switched telephone networks. An entity within the network manages the available bandwidth and executes *admission control*. The advantage of this approach is its simplicity and easier modelling, but it does not use the network as efficiently as possible in case some users do not use all their reserved bandwidth.

The counterpart to that is a *reactive* approach, where the sender adjusts its rate based on feedback from the network. This method allows for far more dynamic regulation and better utilization of resources, better suited for the lively environment of computer networks, which explains why congestion control in TCP has been predominantly reactive. The price of this is complexity, both in designing such a scheme and implementing it correctly. An additional danger is that the coexistence of several variations of such mechanisms within the network might prove to be fatal for some of them, as the unexpected behaviour of their unknown peers will cause them to not perform their function correctly.

An alternative dichotomy is to categorize the mechanisms as *open loop* and *closed loop*, which is the view taken by the field of control theory. Open loop controls use no feedback, which in the case of computer networks means using a priori knowledge about the available bandwidth. Closed loop controls use feedback to determine the action to be taken. We can see that these two categories roughly correspond to the proactive and reactive groups.

These two groups are however not mutually exclusive — one can use a combination of both. This is important for computer networks, because the rate at which the system changes means achieving satisfactory efficiency using only a proactive approach would be extremely difficult.

A network flow starts at the *sender*, goes through zero or more *intermediate nodes* and ends at the *receiver*. From a control point of view, there exist two groups: *controllers*, which can take some action, and *measuring points*, which are able to measure some information. Intermediate nodes can belong to both groups, whereas senders are exclusively controllers and receivers are purely measuring points. A congestion control mechanism could therefore theoretically involve only intermediate nodes; however, this is not very practical. In a scenario involving only a sender and a receiver, the receiver would have no way to inform the sender to adjust its rate. While congestion might not be a problem, in this case, the receiver should be able to protect itself from being overwhelmed by incoming traffic. This is called *flow control*.

Congestion control and flow control are very closely related — the first protects the network, whereas the latter protects the receiver — and both achieve their goals by tuning the flow rate based on received feedback. A sender should take into account both and choose the lower rate that resulted from the corresponding calculations.

2.2.1 Implicit Feedback

Measuring useful metric and propagating feedback back to the sender brings with itself its own set of problems and further increases complexity. Therefore, many algorithms try to work with only implicit feedback.

As we have already seen, when a link becomes congested, queues begin to grow resulting in higher delays and dropped packets. These are two metrics that the sender can use to try to estimate whether congestion is occurring and action needs to be taken. The problematic part is that both delay and packet loss can happen for a variety of reasons and do not necessarily mean that congestion is happening. Using implicit feedback means making assumptions about how the network works. As we see more and more diversity in the underlying networking technologies, such assumptions become more difficult to make — consider for example WiFi networks, which have a higher packet loss rate than wired Ethernet based on their inherent properties, in this case, interference and noise, as well as collisions. Moreover, using packet loss as a sign of congestion means that congestion first needs to take place for the mechanism to act, which is not optimal.

2.2.2 Explicit Feedback

The alternative to implicit feedback is to send feedback that is explicit. This simplifies decision making for the sender, as it is no longer required to guess and make assumptions about the situation.

An example of such approach is called a *choke packet*. When an intermediate node detects congestion, it sends a packet to the sender informing it that it should reduce its sending rate. The benefit of this is that the information is precise, since it is generated by the entity where congestion is happening, and it is fast, in the sense that the sender does not need to wait for any timers to expire to consider a packet lost. However, there are some not so obvious downsides. Creating a notification packet means additional computational effort for the node, especially when it is overloaded. Additionally, it sends more packets into the already congested network, although in reverse direction.

2.3 Additional Factors to Consider

2.3.1 Stability

Feedback and action are not synchronous in the network; there is a finite delay between them. Moreover, the system is noisy, and the observed parameters might get corrupted in transit.[17] A controller should only act on feedback that reflects the system's state.[34] It should react neither too fast, as to let the system stabilize after action was taken, nor too slow, for then the information that prompted the change is outdated.

2.3.2 Scalability

The Internet is constantly growing, as more and more devices become connected. A crucial aspect of any network technology is how well it scales and whether it can keep up with the network expanding. Consider for example keeping per-flow state information. Since the inception of the Internet, link speeds have risen several orders of magnitude, and so has the number of flows a core router would need to keep track of. In the past 10 years, the number of BGP RIB entries in core routers has more than doubled.[13]

2.3.3 Fairness

Last but not least, any congestion control mechanism should be fair regarding the utilization of bandwidth by active users. The naive approach would be to divide the bandwidth equally between participants. However, doing this does not work well because they might be limited by a bottleneck along the path, leading to unused bandwidth. This can be amended by the *progressive filling algorithm*, where each participant progressively gets more bandwidth until a bottleneck is reached. At this point, only the unaffected users continue being given more bandwidth, until all bandwidth is allocated.

Chapter 3

Current Congestion Control Mechanisms

In this chapter, we will look at some of the existing congestion control mechanisms that are employed on the Internet. The first is TCP Tahoe, which was chosen for mostly historical reasons to see what a basic[1] congestion control algorithm might look like. As congestion control and reliable transfer are tightly interconnected in TCP, the section about Tahoe will describe that as well. The next chosen algorithm is TCP CUBIC, which is currently the default algorithm of the Linux kernel, as an example of a more modern but stable algorithm. Afterwards, we move on to more experimental efforts, namely QUIC and TCP BBR, which are both developed by Google. QUIC is unique in this list in that it is not just a congestion control algorithm but a whole transport layer protocol with a specific use case in mind.

3.1 TCP Tahoe

TCP Tahoe is TCP as defined in RFC 1122 [1] and first appeared in version 4.3 of the BSD operating system [20]. TCP itself was first specified in RFC 793 [25]. However, this document does not include any congestion control mechanisms.

TCP achieves reliable communication using retransmission of lost packets. For this purpose, it uses a concept known as the sliding window, which can be seen in figure 3.1. All the data between the *left window edge* and the *right window edge* can be sent into the network. Everything left of the left window edge has been acknowledged as successfully delivered, everything right to the right window edge is waiting to be sent.

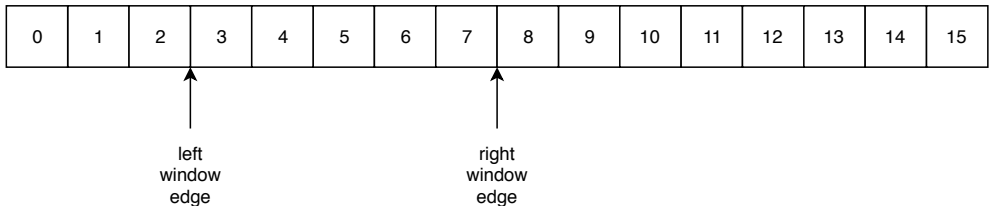

Figure 3.1: A sliding window

[1] It might often be easier to describe a newer algorithm by showing what it does differently from Tahoe. This is in no way meant to demean Tahoe, quite the contrary, given its historical significance.

Moving the left and right window edge is a matter of congestion and flow control. The left edge moves to the right whenever data directly to the right has been successfully acknowledged; moving or not moving the right edge is done by the aforementioned controls and is the mechanism by which the receiver can influence the rate at which the sender transmits data over the network. In practice, this is done by the acknowledgment also carrying information about the advertised window size.

Retransmission of lost data happens after a dynamically computed value called the *retransmission timeout*. It is calculated from the *round trip time* using *exponentially weighted moving average*. First, a *smoothed round trip time* is calculated, which is used to compute the RTO as such:

$$SRTT = (1 - \alpha) * SRTT + \alpha * RTT$$

$$RTO = \min(UBOUND, \max(LBOUND, \beta * SRTT))$$

where $UBOUND$ is the upper bound for the timeout, $LBOUND$ is the lower bound, α is the smoothing factor and β is a delay variance factor.

How these values are computed is important for us because TCP Tahoe uses packet loss to estimate congestion. The original paper by Van Jacobson [15] describes five algorithms for congestion control, but we will focus on two of them — *slow start* and *congestion avoidance*.

In the paper, Van Jacobson bases his work on a principle called *conservation of packets*, which he describes as follows:

> By 'conservation of packets' I mean that for a connection 'in equilibrium', i.e., running stably with a full window of data in transit, the packet flow is what a physicist would call 'conservative': A new packet isn't put into the network until an old packet leaves. The physics of flow predicts that systems with this property should be robust in the face of congestion. Observation of the Internet suggests that it was not particularly robust. Why the discrepancy? There are only three ways for packet conservation to fail:
>
> 1. The connection doesn't get to equilibrium,
>   or
> 2. a sender injects a new packet before an old packet has exited,
>   or
> 3. the equilibrium can't be reached because of resource limits along the path.

3.1.1 Reaching the Equilibrium

Slow start is the initial phase of congestion control. The name is misleading because its purpose is to expand the window to a reasonably large size in a short period of time. The conservation of packets principle states that a packet has to leave the network before another can be put in — one way to look at this is to see the system as self-clocking. The sender uses acknowledgments as a clock for transmitting more packets. A benefit of this is that it adjusts well to bandwidth and delay variations; however, it is difficult to get going. Packets can only be sent as fast as acknowledgments are received, and acknowledgments are limited by the rate of incoming packets.

The slow start algorithm was developed to start this clock by gradually increasing the amount of data that is being transmitted. It consists of making several adjustments to the

sender. A new window called the *congestion window* (*cwnd*) is introduced. The size of this window is set to one packet whenever the connection is starting or restarting. With each received acknowledgment, this window increases its size by one packet. The sender takes the minimum of this and the advertised window when deciding how much data it is allowed to send.

3.1.2 Adapting to the Path

The second violation happens because of wrong retransmission timers, which have already been discussed, so let's now focus on the the third violation. As we've discussed in chapter 2, choosing how to get information about congestion and how to react to it are two main questions of every congestion control algorithm. In this case, a decision was done to use packet loss as a signal that the network is congested, which is a form of implicit feedback. The actions the sender should take are as follows:

1. On a packet timeout, halve the size of *cwnd*.
2. On each acknowledgment, increase the size of *cwnd* by $1/cwnd$.
3. The minimum of *cwnd* and the advertised receiver window should be used when sending.

Slow start and congestion avoidance are commonly used together to form a complete congestion control mechanism. The decision of which algorithm to use is done by introducing a new threshold variable, *ssthresh*. This is initially set to an arbitrary value and is reduced to half whenever a timeout occurs. The size of *cwnd* is compared to *ssthresh* — if smaller, slow start is used, else the window increases according to congestion avoidance.

3.2 TCP CUBIC

TCP is known to under-utilize the available bandwidth, especially in the case of networks with a large bandwidth-delay product (also called *long fat pipes*) [24, 33]. This is caused by the fact that the growth of the congestion window is tied to the round trip time, which means that for long fat pipes, the window could be large but does not grow fast enough. This has led to the development of many new congestion control algorithms designed to improve the situation — one such variant is TCP CUBIC [10].

3.2.1 TCP BIC

CUBIC builds on the ideas of TCP BIC, which uses a different window growth function than what was described for TCP Tahoe [35]. BIC uses two congestion window growth policies — *binary search increase* and *additive increase*.

Binary search increase views congestion control as a searching problem with a simple yes/no feedback. Two variables are used, W_{max} and W_{min} — W_{max} is initially set to the window size just before a packet loss occurred while W_{min} is the size after recovery. The midpoint of these two sizes is computed, which becomes the new congestion window. If the loss occurs, this becomes the new W_{max}, else it becomes the new W_{min} and the process repeats, until the difference becomes less than a *minimum increment threshold* S_{min}.

To prevent the window from growing too fast in a single step, it uses additive increase. If the midpoint is further than the *maximum increment threshold* S_{max}, the window size

is incremented by S_{max} instead. If the window size grows past W_{max}, the algorithm must probe for a new maximum at which packet loss occurs. If the current window size is less than $W_{max} + S_{max}$, increments are done in multiples of S_{min}, starting at $1 * S_{min}$ and doubling it with each step. After reaching $W_{max} + S_{max}$, it switches to the aforementioned additive increase, which is incrementing the window size by S_{max}.

3.2.2 Approximating BIC

As BIC's window growth function is quite complex, CUBIC uses a function that approximates the shape of the BIC growth function. It is also defined in real time and is thus independent of round trip time — this guarantees *RTT fairness*. The window sizes of different flows with varied RTT will increase at the same rate. The function determining the size of the congestion window is defined as such:

$$W_{cubic} = C(t - K)^3 + W_{max}$$

where W_{max} has the same meaning as in BIC, C is a scaling factor, t is the time that has elapsed since last window reduction and $K = \sqrt[3]{W_{max}\beta/C}$ where β is a constant multiplication decrease factor (window reduces to βW_{max}). Growth is also bounded by no more than S_{max} per second to enhance fairness and stability.

In networks with short RTT, CUBIC will grow slower than the RTT dependent TCP. To alleviate this, CUBIC introduces a TCP mode in which the window size after a loss event is computed like this:

$$W_{tcp} = W_{max} \times \beta + 3 \times \frac{1 - \beta}{1 + \beta} \times \frac{t}{RTT}$$

In case W_{tcp} is greater than W_{cubic}, it is used as the congestion window size.

3.2.3 Properties of CUBIC

CUBIC achieves intra-protocol fairness by the use of the multiplicative factor β; a flow with a larger W_{max} will shrink its window more, and since K is proportional to W_{max}, it will grow its window more slowly. Two CUBIC flows will therefore eventually converge to windows of the same size.

Thanks to its independence of RTT for window growth, it is also RTT fair. The growth function is dominated by t — in the case of congestion, any competing flows will have approximately the same t.

The cubic course of the growth function results in longer congestion epochs; however, in contrast with the linear increase in TCP, where long epochs would mean slower growth, it approaches the equilibrium quickly and then stays there for an extended period. A larger portion of the epoch is spent closer to the knee, resulting in better bandwidth utilization.

3.3 QUIC

QUIC is a modern transport layer protocol designed to address some shortcomings of SPDY[30], another Google effort, which later became the basis for HTTP/2 [27, 14]. One of SPDY's features is multiplexing several application streams into a single TCP stream. This results in head-of-line blocking when one of the streams experiences packet loss, as the other streams have to wait until the lost packet is retransmitted. Additionally, shrinking the

congestion window affects all streams, resulting in worse performance overall. Therefore, the motivation is to create a TCP-friendly transport protocol over UDP which would be able to deal with these issues. It further includes features such as forward error correction not commonly found in TCP. The project homepage describes it as „TCP+TLS+HTTP/2 implemented on top of UDP“.[29] As the scope of the QUIC project is quite wide, we will only focus on parts related to congestion control.

Support for pluggable congestion control is built into the protocol; the current default algorithm is a reimplementation of TCP CUBIC.[11] QUIC additionally supports richer signaling than pure TCP — each packet carries a different sequence number, allowing to distinguish between original and retransmitted packets (and their acknowledgments). Furthermore, the delay between reception of a packet and sending the corresponding acknowledgment is carried explicitly in the packet which aids in round trip time calculations. Another feature is the support for distinct stream-level and connection-level flow control to manage the sending rate of the multiplexed streams.

Evaluation of QUIC shows that it generally performs better than TCP CUBIC under loss, by avoiding head of line blocking and recovering the congestion window faster, and in environments with a high delay due to its 0-RTT connection establishment.[16] However, it performs significantly worse in variable delay scenarios, as it incorrectly interprets a large number of reordered packets as packet loss. Although both protocols use the CUBIC algorithm, QUIC increases its window more aggressively and more often; this results in QUIC being unfair to TCP flows by commonly consuming half of all available bandwidth. However, it is fair when sharing bandwidth among several QUIC flows, which is to be expected from a CUBIC algorithm.

3.4 TCP BBR

The final algorithm we will look at is TCP BBR, also developed by Google. The motivation was to improve transmission performance over long fat pipes, for which the traditional loss-based algorithms do not scale well.[3] Instead of interpreting lost packets as congestion, BBR tries to estimate the connection bottleneck and match the sending rate accordingly.

This is achieved by repeatedly measuring the round trip propagation time $RTprop$ and bottleneck bandwidth $BtlBw$. However, only one can be reliably measured at the same time. If the inflight traffic is not greater than BDP (*bandwidth-delay product*, $Bandwidth \times Delay$), more data could still be sent before exhausting the bottleneck bandwidth — but by saturating the bottleneck, queues start to fill which reflects on the round trip time measurements. It is necessary to repeat these measurements, as the values might not be constant durnig the lifetime of a connection — changes in the path can affect both the round trip time (as the pipe becomes shorter or longer) as well as the bottleneck bandwidth (by a wireless link changing its rate). Note that these two parameters can change independently of one another, which is why knowing both is necessary to match the sending rate.

Round trip time can be calculated at time t as:

$$RTT_t = RTprop_t + \eta_t$$

where η stands for additional delays introduced by queues, receiver ACK strategies and other variable factors. By writing round trip time in this way, we can see that $RTprop$ is a physical property of the path that changes only when the path itself changes. As path

changes do not occur very often, $RTprop$ at time T can be estimated as:

$$\widehat{RTprop} = RTprop + \min(\eta_t) = \min(RTT_t) \quad \forall t \in [T - W_R, T]$$

which is a running minimum over the time window W_R. To calculate the bottleneck bandwidth, the delivery rate $deliveryRate = \frac{\Delta delivered}{\Delta t}$ is used:

$$\widehat{BtlBw} = max(deliveryRate_t) \quad \forall t \in [T - W_B, T]$$

where W_B is a time window typically six to ten RTTs.

As learning both values during the same time window is impossible, BBR keeps track of what state it is currently operating in, what can be learned, and how to switch states in case information becomes stale. Most of the time is spent with one BDP in transit, which results in low delay (as delays are caused by queues building up) and gives the ability to learn $RTprop$. Periodically, it increases its sending rate for one $RTprop$ interval to probe for new $BtlBw$; this fills the queues, which are left to drain by spending another $RTprop$ interval sending at a slightly lower rate. For situations when the application does not have enough data to send and the connection is app limited, samples are discarded as they would lead to underestimating the bottleneck bandwidth. Samples are marked as application limited when there was some moment during the time window when there was no data to send.[5, 4]

Experiments done with BBR show that behaviour on links in the Mbps range is fairly consistent with behaviour on links in the Gbps range and that it successfully achieves the goals of reduced queuing delay and reduced loss rate with shallow buffers.[28] Despite that it is shown to be RTT unfair, converging to a fair share only within groups with similar RTTs – the groups themselves are unfair among each other. Another result of the experiment is the confirmation of formal analysis done in [12] that when multiple flows compete, BBR overestimates the bottleneck by about 1.5 BDP. If the buffers cannot hold this additional traffic, the connection will suffer higher retransmission rate than CUBIC, as it does not interpret packet loss as congestion, by a factor of 10. Fairness between a BBR and CUBIC flow is dependent mainly on buffer sizes, but up to 3 BDP, they reach a fair share. However, with multiple CUBIC flows, CUBIC is suppressed and BBR consistently claims more bandwidth.

Chapter 4

RINA

RINA is an architecture whose principles were first outlined in the book *Patterns in Network Architecture: A return to Fundamentals* by John Day [6, 32]. It is an attempt to learn from the lessons and failures of the network technologies since the original ARPANET to try to take a more scientific, less artisan approach to designing network architectures. The most significant insights are that: (I) network communication is interprocess communication (IPC) and nothing but IPC; (II) there is only one layer which repeats as many times as needed. The consequence of these statements is an architecture that is, in many ways, vastly different from the OSI or TCP/IP model.

4.1 Application Processes and Entities

To properly understand RINA and its IPC approach to networking, we will start by looking at the application. The Application Process (AP) is defined as „The instantiation of a program executing in a processing system intended to accomplish some purpose. An Application Process contains one or more tasks or Application-Entities, as well as functions for managing the resources (processor, storage, and IPC) allocated to this AP“ [26]. In turn, the Application Entity (AE) is „a task within an application process directly involved with exchanging application information with other APs“ [26]. In other words, each AP, besides parts to accomplish its primary goal, may contain several AEs which are the parts concerned with communication.

There is only one application protocol called the Common Distributed Application Protocol (CDAP). This protocol allows APs to execute 6 basic operations on objects — *read/write*, *create/delete* and *start/stop*. Anything that an application might need to access is an object outside of the protocol, resulting in a straightforward and unifying approach without having to create specialized protocols. Another consequence of this design is that all application protocols are stateless, as all state is in the application itself. Since each AP can have multiple AEs, they can all expose access to different objects with distinct access control.

4.2 Distributed IPC Facility

Two processes in an operating system which desire to do IPC require the availability of certain functions from the operating system, such as the ability to locate the other process, determine permissions and pass information. Similarly, two APs which want to commu-

nicate over the network require the services of a Distributed IPC Facility, referred to as a DIF. A DIF is similar to a layer in that it is an organizing structure, but the functions which constitute a DIF make it a fundamentally different entity. A DIF is a collection of IPC processes (IPCP) which each execute routing, transport and management functions. Each (N)-level DIF utilizes the services of the underlying (N-1)-level DIF, with the lowest DIF being the physical link itself.

This yields an architecture that scales indefinitely, with any constraints imposed not being a property of the architecture itself but physical limitations. All DIFs have the same functions, but with different scope and range. In practice, this means that for example routing is done several times at one node, and there are several routing tables, each with different information.

A DIF is a specific case of a Distributed-Application Facility (DAF) where the application is providing IPC. Each IPCP is therefore an AP as well. Generally, a DAF is „a collection of two or more cooperating APs in one or more processing systems, which exchange information using IPC and maintain shared state. In some Distributed Applications, all members will be the same, i.e., a homogeneous DAF, or may be different, a heterogeneous DAF“ [26].

Membership in DIFs is dynamic — hosts may join and leave DIFs as needed, or even create a new DIF if no suitable one exists, provided they share some (N-1)-DIF with the neighbours they want to communicate with. As the IPC a DIF provides in itself an application, each DIF also holds some state, which includes various attributes related to how the DIF operates. An example of such might be the authentication policy. In the case that host A and B who are enrolled in the same DIF require stronger authentication than the DIF provides, they can establish a new (N+1)-DIF with the desired policy over the common (N)-level DIF.

4.3 Addressing

Due to the recursive nature of the architecture, addressing is noticeably different from the TCP/IP model. First of all, there is no global address space — each address is a synonym for an IPCP local to the DIF it appears in. All layers route on their addresses, and while there exists a (N) to (N-1) address mapping, this routing is done by the (N-1) layer. Addresses are structured to facilitate their use within the layer, i.e., they might contain topological information.

While this may appear deceptively simple, a number of useful properties come out from this structure. Chief among them is support for multihoming and its extreme case, mobility, which require a lot of work to function properly in TCP/IP. As the routing vertically between DIFs is done in the lower DIF, consequently multihoming is simply an IPCP having multiple (N-1) bindings. Mobility can be viewed as nothing more than multihoming with the (N-1) bindings appearing and disappearing more frequently. (N) addresses may change by joining a different DIF (as they are of local significance) in which case it is simply a matter of creating a new binding; or because its topological significance changes, where the process is more complex. The IPCP receives a new synonym and starts using it for all ongoing communication, prompting the other side to switch to using the new address as well. As all communication is transferred to this new address, the old one dies out.

The second significant difference is RINA's support for Application Process Names (APN), as a globally unambiguous name for the AP, unlike TCP/IP where applications are

identified by a combination of the host's IP address and the (TCP/UDP) port number. To make matters worse, the port number is overloaded with three different semantics — the application name, the port-id for the application to communicate with the flow (the socket) and as a connection endpoint. RINA decouples these intro three separate entities, with the port-id having only local significance for the AP and the Connection-Endpoint-id (CEP-id) being local to the AE it originated in. The combination of the source and destination CEP-ids creates (together with the QoS-id) a connection-id, which appears in the PDUs sent over the wire.

4.4 Transport Protocols

Mechanisms are invariant parts of the protocol that are ever-present, whereas *policies* are variant parts that can be negotiated during the communication establishment phase depending on the higher-level requirements. Properly separating protocol components into these two categories reveals that there is no need to for multiple different transport protocols; the existence of only one is perfectly sufficient.

To provide an example, consider the requirements for media streaming traffic. Optimally, the PDUs would be delivered in-order, but there is no need for reliable traffic — in fact, reliable delivery is undesirable. In the current Internet, the application has a choice between TCP and UDP, with neither being fully sufficient — UDP does not do in-order delivery while TCP imposes reliable transfer. The process of acknowledging PDUs is rigidly built into TCP. With a better-designed protocol, *how acknowledgment works* would be a mechanism but *when to acknowledge* would be a policy. In the case being considered, the policy could be to always acknowledge PDUs, even if they did not arrive. This approach also shines some light on the actual semantics of ACKs — they do not mean *I have received the PDU*, as it is commonly understood, but *You can send more PDUs*; a slight yet significant difference. [6]

There is another substantial dichotomy related to the operation of transport protocols which needs to be contemplated to understand the reasoning leading to the design of protocols in RINA. We can separate mechanism into two groups based on how tied they are to the PDUs being transferred — *tightly-coupled* and *loosely-coupled*. Tightly-coupled mechanisms must be associated with every PDU and handle fundamental aspects of data transfer, such as integrity checks and sequence numbering. The loosely-bound mechanism, on the other hand, relates to features associated with the data transfer in general, namely flow control and reliability. It is possible to include another group, consisting of even more loosely-coupled elements required for data transfer, such as routing and resource allocation.

These groups each operate at a different timescale and frequency, the most tightly-coupled working the fastest and the least tightly-coupled the slowest. At the same time, the computation required grows in complexity, being fairly straightforward for the former and rather complicated for the latter. They are decoupled from each other by a shared state vector which holds the state information. Tightly bound mechanisms write to this state vector, whereas loosely-bound mechanisms read from it.

Looking at this structure, it becomes clear that there are two protocols involved with data transfer, each with its Protocol (state) Machine (PM) and bound by the state vector. In RINA these protocols are called the Data Transfer Protocol (DTP) and Data Transfer Control Protocol (DTCP), and together they form the Error and Flow Control Protocol (EFCP).

The design of DTCP is based on Watson's Delta-t protocol [7]. As his insights into the nature of transfer protocols have shown, all properly designed data transfer protocols are soft-state. Maintaining explicit state synchronization using SYNs and FINs is unnecessary; time-based synchronization is sufficient for reliable data transfer. All that is required for reliable communication is the existence of three timers with upper bounds: (I) Maximum Packet Lifetime (MPL), which denotes the maximum amount of time that a packet can exist in a network; (II) a retransmission timer which denotes how long the sender stores an unacknowledged message with the ability to retransmit it; (III) an acknowledgment timer which denotes how long the receiver can wait before sending an acknowledgment of a received PDU. Delta-t assumes that all connections exist all the time, but synchronization state is maintained only for the duration of data transfer activity. After periods of inactivity, the state may be discarded.

4.5 IPCP Components

The purpose of the IPCP is to allow the system to do IPC with other DIF members. To achieve this goal, it performs a number of functions which could be categorized into three groups analogous to ones described in the previous section. The functions and their categories are as follows:

Data Transfer

- Delimiting
- Data Transfer
- SDU Protection
- Relaying and Multiplexing Task

Data Transfer Control

- Error Control
- Flow Control

Management

- Enrollment
- Flow Allocator
- Resource Allocator
- Common Distributed Application Protocol

Detailing each component's purpose and operation is vastly out of the scope of this thesis; this listing mainly serves as a reference to give the reader an idea of what parts the IPCP actually consists of.

4.6 IPC Communication

This section gives a step-by-step overview of how communication in RINA works, going through all the necessary procedures that need to be executed before data transfer can happen. The goal of this is to put all of the pieces together and make the reader familiar with the process, as this knowledge will be useful when discussing the proposed flow allocation extensions.

4.6.1 Enrollment

For two IPCPs to establish a communication channel, they need to be members of the same DIF. If there is no DIF, these two IPCPs have in common, one must join a DIF the other one is already a member of.

The process of joining a DIF is commonly referred to as *enrollment*. Given that IPCP i wants to communicate with IPCP k in DIF A, it first needs to share some DIF B with an IPCP j, with j also being a member of A and B being an underlying DIF of A, as seen in figure 4.1. Using the common DIF B, i requests that an IPC channel be established with j using j's Application Process Name.

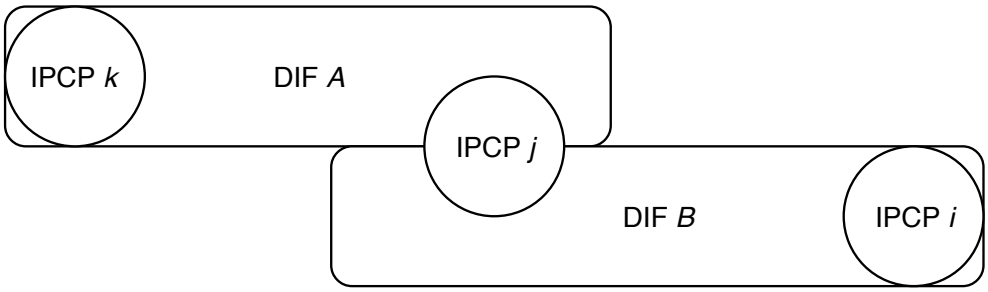

Figure 4.1: Overlaying DIFs for enrollment

After this connection has been established, j authenticates i, with the strength of the authentication depending on the DIF A requirements, and determines whether i can access j. If i is allowed to access A, it gets assigned an address (meaningful only within the DIF A) and initialization parameters associated with A are shared with i.

4.6.2 Flow Allocation

The next step in establishing a communication channel is allocating a flow between the two AP instances. i requests service using k's APN. The local IPCP returns a port-id to the AE, with only local significance. A set of EFCP policies is chosen based on the request, and an EFCP instance is created, identified by a CEP-id. The local IPCP continues by creating a request to find the destination application. When the destination IPCP receives the request, it decides whether to accept or deny the request, depending on the access control policy. If the request passes the necessary criteria to be accepted, an EFCP instance is created, with a CEP-id, and this result is sent to the requesting IPCP. Both CEP-ids are concatenated to be used as a flow identifier. After the local IPCP receives the positive response, a binding is created between the port-id and the CEP-id.

Chapter 5

The Raft Consensus Algorithm

As the name itself suggests, the basic building block of RINA — the *Distributed IPC Facility* — is a distributed system. Each IPCP in the DIF contains a *Resource Information Base* (RIB) to store various information necessary for operation. It is important that the state of this RIB is consistent and coherent across the system, so that each IPCP operates with same information.

This problem is known as *consensus* within a distributed system. Consensus algorithms ensure that nodes reach agreement on shared state even in case of failures [22]. The most well known algorithm is Paxos [19], used in implementations such as Chubby [2]. Rlite uses a newer algorithm called Raft to implement distributed consensus.

Raft [23, 22] was designed with *understandability* as one of its primary goals. One of the chief criticisms raised against Paxos was its complexity and difficulty in adapting to real-world systems. Raft attempts to be both easily understandable and easily usable in practice. It ensures that the state of replicated state machines is consistent across the distributed system.

5.1 The structure of Raft

Raft achieves understandability by decomposing consensus into three separate subproblems — *leader election*, *log replication* and *safety*. First, a leader is elected from among the participating nodes. The leader is responsible for accepting log entries from clients and replicating them on other nodes, as well as committing said entries to the replicated state machines. The strong function of a leader simplifies the decision making and data flow in the system. In the case of leader's failure, a new leader is elected and the process continues.

Each node in the system is in one of three possible states — *leader*, *follower* or a *candidate*. During normal operation, exactly one leader exists; the rest are passive followers who only respond to requests from leaders and candidates. The leader handles all client requests and manages log replication. Candidate is a temporary state used during the election process.

Raft uses a concept of *terms*. A term is a period of time of arbitrary length. Terms are numbered sequentially with integers, and act as a logical clock. Each term begins with an election, and the winning candidate takes the position of the leader for the rest of the term. It is possible for an election to end without a leader — e.g. due to a split vote — in which case the term ends without a leader and a new term with a new election starts.

Each node keeps track of the current term number. This number is included in the messages exchanged between nodes. If a follower receives a message with a higher term, it updates its own value. A candidate or a leader discovering a newer term immediately revert to a follower state.

5.1.1 Leader election

Leader election is the first phase of a system. Its purpose is to agree on a leader that will serve a distinguished function for the rest of a term.

A hearbeat mechanism is used to trigger an election. Each node initially starts in the follower state, and remains so as long as it keeps receiving a heartbeat from a leader. If it does not receive a heartbeat notification for longer than the *election timeout*, it assumes there is no active leader and begins an election.

The follower initiating an election first increments its term number. It then votes for itself and asks for votes from the other nodes in the system. It remains in this state until a) it wins the election b) another node announces itself as a leader or c) there is no winner for a certain period of time.

To win an election, a candidate must receive a majority of the votes in the term. Each node may only vote for one candidate, and votes are given on a first-come-first-serve basis. Requiring a majority vote ensures that at most one node may win an election in any given term. After winning a vote, the follower assumes the position of a leader and begins sending heartbeat messages to the rest of the nodes.

The candidate may receive a message from a node claiming to be a leader. If the leader's term number is at least as high as the current term, it recognizes the leader as valid and ends the election by becoming a follower. If the term number is lower, the message is ignored.

It is possible for no candidate to win the election, due to the votes being split in such a way that no obtains a majority. In this case, the election times out and a new one (with a new term number) begins. This timeout is chosen randomly from a fixed interval, to ensure that split votes are rare and do not repeat indefinitely.

5.1.2 Log replication

Once elected, the leader starts receiving request from the clients and servicing them. The requests contain commands to be executed by the replicated state machines. When the leader receives a command, it appends it to its log and distributes it to all followers. The entry is applied only after it has been successfully replicated across the system, and the result of the execution is returned to the client. The leader retries distributing the long entry until all nodes have successfully stored it.

Each log entry contains the command to be executed and the term number when it was received, to help detect inconsistencies between logs.

The leader is in charge of deciding when to apply — *commit* — an entry from the log to the replicated state machines. Raft guarantees that each committed entry will eventually be executed by all available state machines. An entry is committed once it has been replicated on a majority of nodes. Committing an entry also commits all previous entries in the log.

The leader keeps track of the index of the newest entry that has been committed, and includes this as part of the heartbeat messages to the followers. Once a follower learns that an entry has been committed, it applies it to its local state machine. In the case of multiple unapplied entries, they are applied in the order they are stored in the log.

Chapter 6

Guaranteeing bandwidth

Rlite is an LGPL licensed implementation of the Recursive InterNetwork Architecture created by Vincenzo Maffione, to which I have contributed in the past [21]. It aims to be a lightweight but performant implementation that can be used in production, with a focus on a clean and simple design.

Conceptually, rlite is split into user and kernel space parts, with several kernel modules implementing various shim DIFs, such as IPCP over Ethernet or over UDP/TCP. The main *rlite* kernel module creates two I/O devices, `/dev/rlite` for management and control and `/dev/rlite-io` for passing data to the kernel. Most of the functionality is done in userspace, in the *Userspace IPCP* daemon (UIPCP). This daemon implements high-level functionality such as flow allocation and routing. The state of the daemon is synchronized with the kernel module, to provide robustness against crashes and failures in the userspace.

In the default implementation, flow allocation always succeeds. The goal, firstly, was to implement support for reserved bandwidth. The idea is reminiscent of the way telephone circuit networks used to work — each customer gets allocated a certain share of the bandwidth that is guaranteed to be available to them for the whole duration of the connection. Whether they fully utilize this bandwidth or not is up to the customer, which necessarily leads to lower overall efficiency than with TCP congestion control, as another customer cannot „borrow“ the bandwidth of one that does not completely saturate their reservation — this would violate the availability guarantee.

The ability to support bandwidth reservations requires support from several components. The *Lower Flow Database*, which contains the (N-1) flows available to the DIF needs to include information about available bandwidth. The *Routing* component must build a graph of the network from the database, and contain a method to find a possible path for the requested flow using a sufficient graph flow algorithm. In addition, new routes need to be injected in the network, which route on more fields than the destination. The flow must be routed along the reserved path (and not the path computed by the general — in this case link-state — algorithm) for the reservation to work properly. The *Flow Allocator* needs to check whether a suitable path exists before continuing with the allocation, and deny the request in the case it cannot be serviced.

As RINA allows composing of DIFs in various ways, it is worth pointing out that this implementation is meant for single DIF scenarios — that is, one N-DIF on top of Ethernet shim (N-1) DIFs. A policy defines the behaviour of a single IPCP, which can only interact with IPCPs in the same DIF. [1] Propagating the bandwidth information to IPCPs in

[1] Communication with (N-1) IPCPs is of course possible through the application API.

higher/lower DIFs would require the support of a *DIF Management System*, which drives the RIBs of all the DIFs in the system, similar to a SDN controller. Such DMS is quite complex and should be independent of a particular RINA implementation, and no such systems exist as of the time of writing.

6.1 Policy dependencies

Rlite includes support for component policies, meaning that multiple implementations (behaviours) can coexist, and can be switched between during runtime.

Basic support for switchable policies has been a part of rlite for some time now, and was one of my first contributions to the project. During the research and design phase of this thesis, it became clear that there will need to be multiple new policies included, for different components, and that those policies need to be used together, else the system becomes inconsistent. As they have hard dependencies on each other, activating one must activate the other.

The policy framework was therefore extended to support dependencies [18]. The call to register a policy was extended to include an optional list of dependencies as a parameter. Declaring one way dependencies is therefore trivial. In some cases, a tighter binding might be required — multiple policies might cyclically depend on each other. To make such cases user friendly, a new call to register a *policy group* was added. It takes a list of policies as an input and creates the cyclical dependencies (all-on-all) for every policy before registering them. The policy group therefore becomes one tightly-coupled unit, where activating one policy from the group will activate all of them. An additional benefit of this is that it makes configuration and management easier, as the administrator needs to only specify the policy they are interested in, and the required supporting policies will be switched automatically.

The actual policy switching mechanism was extended to handle dependencies in a safe and predictable manner, to ensure system stability. Firstly, a a policy list is built, starting with the requested policy, by traversing the dependency tree using BFS. Each new dependency found is first checked to be available (registered in the system) and appended to the list, while dependencies already in the list are not included again. If a declared dependency is not found (is not registered as available in the system), the whole policy switch operation fails, before any policy switching occurred. This means that the switch operation works in a transaction-like manner, where it either succeeds in full, or fails without making any changes, to avoid bringing the system into an inconsistent state. When all the dependencies have been found, the operation continues with the actual switching.

Policies are switched in reverse order —that is, the requested policy is activated last —to ensure the required dependencies are already in place when switching. The list is traversed, and each policy is activated. If a policy is already active, no action is done —it is simply skipped over.

This extension of the policy framework allows us to create more complex policies, which utilized more than one component, while keeping the configuration and management simple and guaranteeing some degree of safety for the operation.

6.2 Getting bandwidth information into the LFDB

The Lower Flow Database — as the name suggests — contains information about the lower (N-1) flows available to the DIF. Each IPCP in the DIF has a copy of the LFDB as part of

the RIB. This is very useful for graph computations in the DIF — the default (link-state) routing policy relies on the LFDB to build the network graph as an input for the Dijkstra shortest path algorithm. However, the information about available bandwidth was not part of the LFDB.

Linux system call `ioctl()` „manipulates the underlying device parameters of special files“ [31]. As almost everything is a (special) file in Linux, it can be used to query various information about the network interface using the `SIOCETHTOOL` command. In this way, we are able to get the speed of a particular interface.

Each DIF on the node gets its own UIPCP process. Apart from *normal* DIFs, rlite includes several shim DIFs. The important one here is the *shim-eth* DIF, as it is usually the lowest DIF in a system, corresponding to the Ethernet link between machines. Therefore, during UIPCP creation, we check whether we are a shim-eth DIF — if true, we query the interface speed and store it as part of the UIPCP information. [2]

The next step is getting the bandwidth information into the LFDB of the (N+1) DIF. This is part of the Routing component. When a new node enrolls into the DIF, the Routing component gets notified about the new node. If it is a neighbour (that is, they share a (N-1) DIF), it creates an entry for the supporting lower flow in the LFDB and propagates it across the DIF. Here, we need to iterate over the supporting flows to see if they belong to a shim-eth DIF. If yes, we look up the corresponding UIPCP and set the flow bandwidth according to the information stored there. As the information about a new lower flow is propagated through the DIF, eventually all nodes will have this information in their LFDB.

6.3 Finding the flow path

Once we have the bandwidth information in the LFDB, we need to be able to find a satisfactory path in the network graph. The LFDB builds the graph as part of the shortest path computation for routing already, so it is easy and convenient to reuse it, focusing on the flow problem instead.

Finding the maximum flow between two nodes is a well known problem, and methods like the Ford-Fulkerson algorithm exist [9].

Algorithm 1 Ford-Fulkerson maximum flow algorithm

Input: a flow network $G = (V, E, s, t, cap)$
Output: a maximum flow f
  Initialize $f(v, w) = 0$ for all $(v, w) \in E$
  Calculate residual network G_f
  while an augumenting path in G_f exists **do**
    Let P be an augumenting path in G_f with capacity Δ_P
    Obtain increased flow f_P using flow-augumenting path P
    $F := f_P$
    Update residual network G_f

[2] This might seem as a contradiction with the information above regarding inter-DIF IPCP communication. The necessary distinction here is that there is no communication taking place, and it is not accessing the contents of the RIB. The (N-1) DIF is simply accessing the list of UIPCPs in the system and taking some (static) metadata from it, similar to the UIPCP name or the DIF it belongs to. Propagating changes of available bandwidth would require some kind of notifications — that is, communication, as well ass accessing the contents of the RIB.

One of the problems with the Ford-Fulkerson algorithm is that picking the augmenting path is not well defined. Modifications such as Edmonds-Karp exist to solve this issue. Edmonds-Karp in particular uses BFS in every iteration to find the augmenting path, which means that each iteration always finds the shortest path possible. This is convenient for our use case, as we would like to prefer shorter paths when possible (just like the link-state routing does).

Algorithm 2 Edmonds-Karp maximum flow algorithm

Input: a flow network $G = (V, E, s, t, cap)$
Output: a maximum flow f

```
repeat
    Let q be a queue
    Push s to q
    Let pred be an array holding the preceding edge on the current path
    while q is not empty do                          ▷ BFS to find augmenting path
        Retrieve w from q
        for all (u, v) ∈ E : u = w do
            if pred[v] = null ∧ (v, u) ≠ pred[u] ∧ cap(u, v) > 0 then
                pred[v] := (u, v)
                Push v to q
    if pred[t] ≠ null then                           ▷ Found and augmenting path
        Δ_f := ∞
        (u, v) := pred[t]
        while (u, v) ≠ null do                       ▷ See how much flow we can send
            Δ_f := min(Δ_f, cap(u, v))
            (u, v) := pred[u]
        (u, v) := pred[t]
        while (u, v) ≠ null do                       ▷ Update the path
            cap(u, v) = cap(u, v) − Δ_f
            (u, v) := pred[u]
        f := f + Δ_f
until pred[t] = null
```

However, the Edmonds-Karp algorithm finds the *maximum flow between the two nodes*, spread over different paths. We are interested in a single (shortest) path that has enough capacity for our desired flow. Therefore we need to modify the algorithm in several ways. First, we change the BFS condition to check whether the capacity is large enough; not that there is any available. This results in getting only augmenting paths that can service the whole flow. As we only need the first augmenting path, as that will be the shortest, we can remove the whole second part of the algorithm. The final change consist in the output of the algorithm — we need to return the found path instead of the maximum flow. We can use the *pred* array to build a list of nodes after we have successfully found the path.

Our resulting algorithm bears more resemblance to BFS than the the original Ford-Fulkerson, in great part due to only requiring a single path. In fact, starting „from scratch“ using a simple BFS and adding the flow condition might have been a satisfactory approach. The running time is linear, specifically $O(2 * E)$ as we iterate over all the edges possibly twice, in the case of a linear graph.

Algorithm 3 Algorithm for finding the shortest path of at least certain capacity

Input: a flow network $G = (V, E, s, t, cap)$ and a required flow R
Output: a viable path P

```
Let found := False
Let q be a queue
Push s to q
Let pred be an array holding the preceding edge on the current path
while not found ∧ q is not empty do                    ▷ Find the shortest viable path
    Retrieve w from q
    for all (u, v) ∈ E : u = w do
        if pred[v] = null ∧ (v, u) ≠ pred[u] ∧ cap(u, v) ≥ R then
            pred[v] := (u, v)
            Push v to q
if pred[t] ≠ null then                                 ▷ Build the path
    Let P := {t}
    for all (u, v) ∈ pred starting from pred[t] do
        Prepend u into P
        Continue to pred[u]
```

A new Routing policy was created, which exposes this method to other components, as it is part of the LFDB (which is contained in the Routing policy). We will require this functionality to be available from the Flow Allocator.

6.4 Distributed flow allocation

The core of the work is the Flow Allocator policy, which manages the creation and destruction of flows. When a flow allocation request is received, and it includes a bandwidth requirement, it needs to postpone the response until a reservation either succeeds or fails. As the reservation process requires the cooperation of multiple nodes in the DIF, it is by nature a distributed process. Rlite contains an implementation of the Raft consensus algorithm, which was described in chapter 5, to implement centralized but fault tolerant policies.

Each IPCP in the DIF can act as a client, to send bandwidth reservation requests. Exactly one distinguished leader exists in the DIF and serves the client requests. In addition, some of the IPCPs act as distributed replicas to provide fault tolerance, and the leader synchronizes its state machine with them. The replicated state machines store the active reservations, and are therefore able to take over in the case of leader failure.

Flow allocation works in four steps:

1. The application requests a flow from the local IPCP.
2. The local IPCP contacts the remote IPCP.
3. The remote IPCP asks the remote application whether to accept the flow.
4. The local IPCP receives the response and returns it to the application.

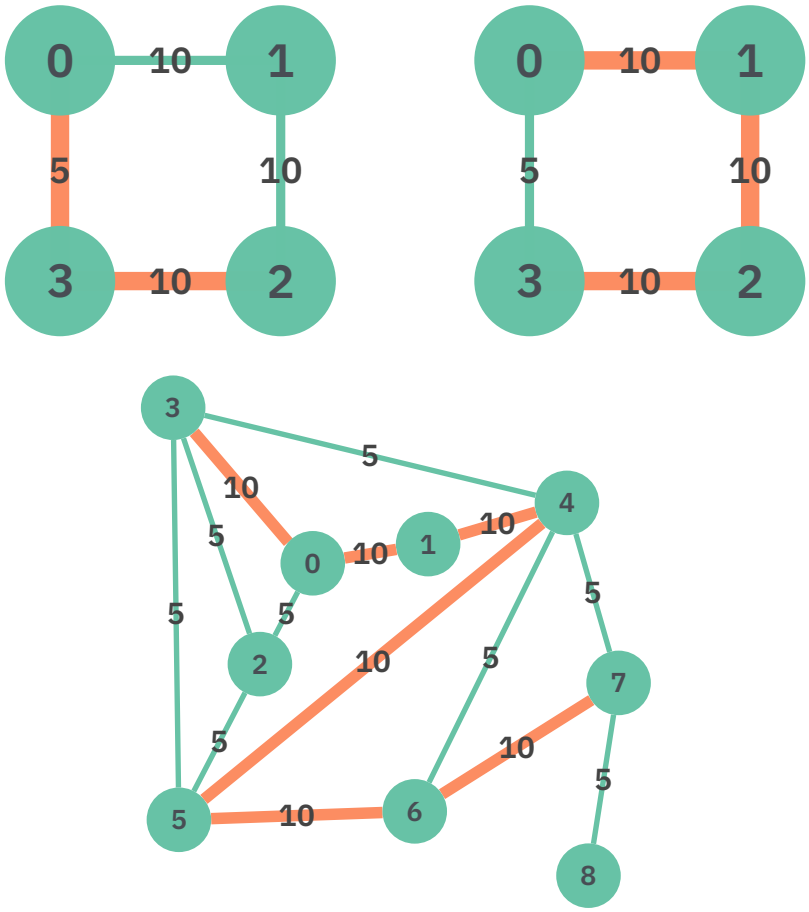

Figure 6.1: Some of the paths found (orange) by the algorithm. In the top left, a flow from 0 to 2 of capacity 4 was requested. In the top right, a flow from 0 to 3 of capacity 7. In the bottom, a flow from 3 to 7 of capacity 7. All of these examples (but not only) are also included as unit tests.

After step 4., a flow has been successfully allocated and communication can begin. As the applications have already agreed to exchange messages, any bandwidth reservation should have already taken place and be ready for use.

Therefore, the flow allocation process with support for bandwidth reservations is as follows:

1. The application requests a flow from the local IPCP.
2. The local IPCP contacts the Raft leader with a request for the bandwidth reservation.
3. The Raft leader forwards the request to the remote IPCP, which accepts or rejects the request.
4. The remote IPCP responds to the Raft leader. If the request was accepted, the leader tries to find find a suitable path. If such a path was found, it appends the reservation command to the replicated log and distributes route updates for the reserved path.
5. The Raft leader forwards the response to the local IPCP.

6. The local IPCP informs the application about the result.

All flow allocation requests therefore go through the Raft leader, which acts as a forwarding proxy between the local and remote IPCP during flow allocation. By receiving each flow request, the Raft leader is able to keep track of all the flows (and reservations) that occur.

The IPCP, in the role of a Raft client, sends the request for flow allocation that would normally be destined for the remote node to the Raft leader. It contains the complete flow request information, as it will be forwarded to the remote node later. The Raft leader requires only some of this information, namely the local IPCP name, remote IPCP name and the local port to create a *Flow-id* uniquely identifying the flow. In addition, it needs the requested bandwidth as input to the graph algorithm.

Once the leader receives the request, it looks for a viable path. If none are found, it sends a negative response to the client, which gets passed back to the application. If a path is found, it creates entries to append to the replicated state machine log. Due to implementation limitations, the size of the entry structure needs to have a fixed size. As we want to pass a (potentially endless) list of nodes making up the path to be reserved, we need to work around this by creating a separate entry for each lower flow (hop). The replicated state machine on the other end uses these to build the path again. A final entry signifies the end of the path, and once received, the bandwidth change along the reserved path is propagated into the LFDB.

Only after this final entry has been committed is the request forwarded to the remote IPCP, and flow allocation continues. The remote IPCP returns a (positive or a negative) response to the Raft leader, which now (in addition to the information passed in the request) includes the remote port and remote CEP-id. It is in this moment the Raft leader is able to distribute the necessary flow routes, as the local and remote CEP-ids are part of the routing entries. While it is possible for one flow to alternate multiple connections, for now the mapping between a flow and a connection is 1 to 1. Routing on port-ids is not possible, as those are not part of the EFCP PDU.

Finally, the Raft leader forwards the response to the local IPCP, which returns the response to the requesting application, and communication can begin.

Flow deallocation, likewise, goes through the Raft leader. The request is sent to the Raft leader, which removes the reservation, and forwards it to the remote IPCP. Afterwards, it deletes the per-flow routing rules.

As not all flows might specify the requested bandwidth, two policy parameters control the behaviour of such flows. The `reject-unlimited-flows` parameter, which takes `true` or `false` values, controls whether flows that do not set the bandwidth field in the request should be rejected. The second parameter, `default-unlimited-bandwidth` specifies the bandwidth that should be set for such flows if they are not rejected.

6.5 Example operation

What follows is a series of outputs from `rlite-ctl dif-rib-show`, which show the information currently stored in the RIB, edited for brevity. It shows the LFDB and the Flow Allocator, showing the known lower flows and their bandwidth, as well as the currently managed flows and active reservations, if any.

```
Lower Flow Database:
    Local: n1.b.IPCP, Remote: n1.c.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.b.IPCP, Remote: n1.a.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.a.IPCP, Remote: n1.b.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.c.IPCP, Remote: n1.b.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.c.IPCP, Remote: n1.d.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.d.IPCP, Remote: n1.c.IPCP, Total: 100 Mbps, Free: 100 Mbps

Supported flows (Src/Dst <Appl,IPCP,port>):

Flow reservation table:
```

Figure 6.2: The RIB after bringing the network up, in an idle state. No flows are active, and all bandwidth is available. No reservations exist.

```
Lower Flow Database:
    Local: n1.b.IPCP, Remote: n1.c.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.b.IPCP, Remote: n1.a.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.a.IPCP, Remote: n1.b.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.c.IPCP, Remote: n1.b.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.c.IPCP, Remote: n1.d.IPCP, Total: 100 Mbps, Free: 100 Mbps
    Local: n1.d.IPCP, Remote: n1.c.IPCP, Total: 100 Mbps, Free: 100 Mbps

Supported flows (Src/Dst <Appl,IPCP,port>):
    [R], Src=<rinaperf-data|client,n1.a.IPCP,2>
         Dst=<rinaperf-data|server,n1.d.IPCP,2>
         Connections: [<SrcCep=1, DstCep=1, QosId=0> ]
    [R], Src=<rinaperf-data|client,n1.a.IPCP,1>
         Dst=<rinaperf-data|server,n1.d.IPCP,1>
         Connections: [<SrcCep=0, DstCep=0, QosId=0> ]

Flow reservation table:
```

Figure 6.3: The RIB while running a throughput test using rinaperf. Two flows are active, but as no bandwidth was requested, no reservations exist, and all bandwidth is available.

```
Lower Flow Database:
    Local: n1.b.IPCP, Remote: n1.c.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.b.IPCP, Remote: n1.a.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.a.IPCP, Remote: n1.b.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.c.IPCP, Remote: n1.b.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.c.IPCP, Remote: n1.d.IPCP, Total: 100 Mbps, Free: 20 Mbps
    Local: n1.d.IPCP, Remote: n1.c.IPCP, Total: 100 Mbps, Free: 20 Mbps

Supported flows (Src/Dst <Appl,IPCP,port>):
    [R], Src=<rinaperf-data|client,n1.a.IPCP,2>
         Dst=<rinaperf-data|server,n1.d.IPCP,2>
         Connections: [<SrcCep=1, DstCep=1, QosId=0> ]
    [R], Src=<rinaperf-data|client,n1.a.IPCP,1>
         Dst=<rinaperf-data|server,n1.d.IPCP,1>
         Connections: [<SrcCep=0, DstCep=0, QosId=0> ]

Flow reservation table:
     n1.a.IPCPn1.d.IPCP2: n1.a.IPCP->n1.d.IPCP
       (n1.a.IPCP,n1.b.IPCP,n1.c.IPCP,n1.d.IPCP,) : 40 Mbps
     n1.a.IPCPn1.d.IPCP1: n1.a.IPCP->n1.d.IPCP
       (n1.a.IPCP,n1.b.IPCP,n1.c.IPCP,n1.d.IPCP,) : 40 Mbps
```

Figure 6.4: The RIB while running another rinaperf test, this time requesting 40Mbps per flow. Again, two flows are active, but the flow reservation table lists two entries, and the available bandwidth has decreased accordingly. After the flow is deallocated, the output is identical to the one shown in figure 6.2 (bandwidth is freed and reservations removed).

Chapter 7

Evaluation

The implemented algorithm is able to utilize the available bandwidth efficiently according to the demands of applications. Each flow that get successfully created is guaranteed to have the necessary bandwidth available. This means that congestion control algorithms should never come into play, as each flow is rate limited to the requested bandwidth, and the total traffic in the system should never exceed the available link capacities.

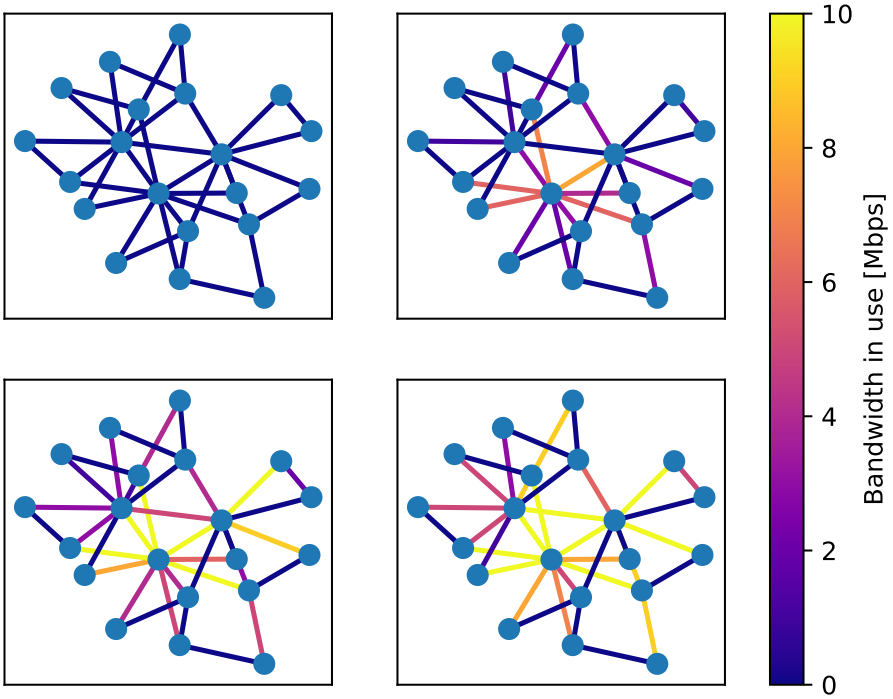

Figure 7.1: Timelapse of network link utilization by allocating flows at random.

Figure 7.1 shows the traffic in flow in a test network. Each link in the network has a capacity of 10 Mbps, and 150 flows were attempted to be allocated from a random source to a random destination, each requesting 1 Mbps. Snapshots were taken at intervals of 50 allocations, the first being before any allocations were done. As it can bee seen from the figure, the links get progressively more and more utilized, especially in the centre and close to the nodes that make up busy „exchange points“ with many links. This agrees with

the intuitive observation that as more flows are created, the backbone of the network will become more strained. We can see that the traffic is distributed quite evenly along the less used edge links.

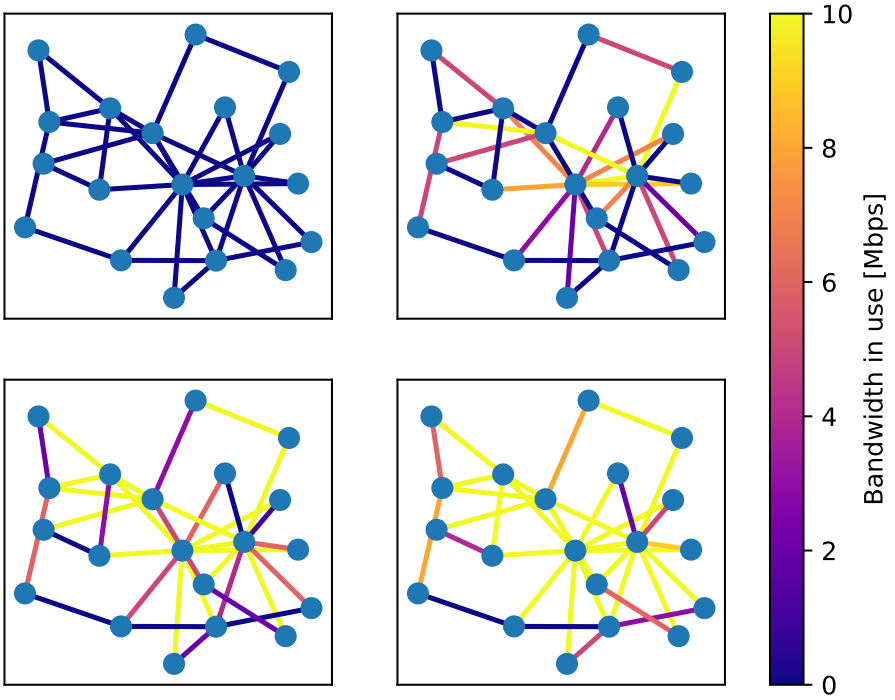

Figure 7.2: Another timelapse of network link utilization by allocating flows at random.

Figure 7.2 shows a similar experiment with a diffrent topology. The snapshot interval was set to 100 flow allocations, which results in even higher overall utilization in the bottom right graph. Again, the overall utilization pattern is similar to the one from figure 7.1, showing that the algorithm works in a fairly predictable manner, even with flow allocations picked completely at random, and working on different network graphs.

The fact that each flow is routed differently is a great benefit that allows for better bandwidth utilization in networks with multiple alternate paths from one end of the network to another. When routing only by destination address, all the traffic to the same destination will go through the same link. In the example network in figure 7.3, only one of the backbone links would get used in a normal scenario. This is why backbone links usually have higher bandwidth requirements, as they are a bottleneck that aggregates the traffic from many different parts of the network. By using per-flow routing, we are able to distribute the the traffic more evenly, which also translates to lower interface speeds being required on these links — multiple slower links are able to handle the traffic just as well. [1]

Although the algorithm is logically centralized, it is distributed and fault tolerant by utilizing the Raft consensus algorithm. Figure 7.4 once again shows the output from the `rlite-ctl dif-rib-show` command, edited for brevity. We can see from the output that the information about the active reservation has been successfully replicated from IPCP

[1] Not considering the scenario where someone makes a reservation which is larger than the bandwidth of the slower interface. However, single flows requesting an interface's worth of bandwidth of more are closer to denial of service attacks than legitimate user behaviour.

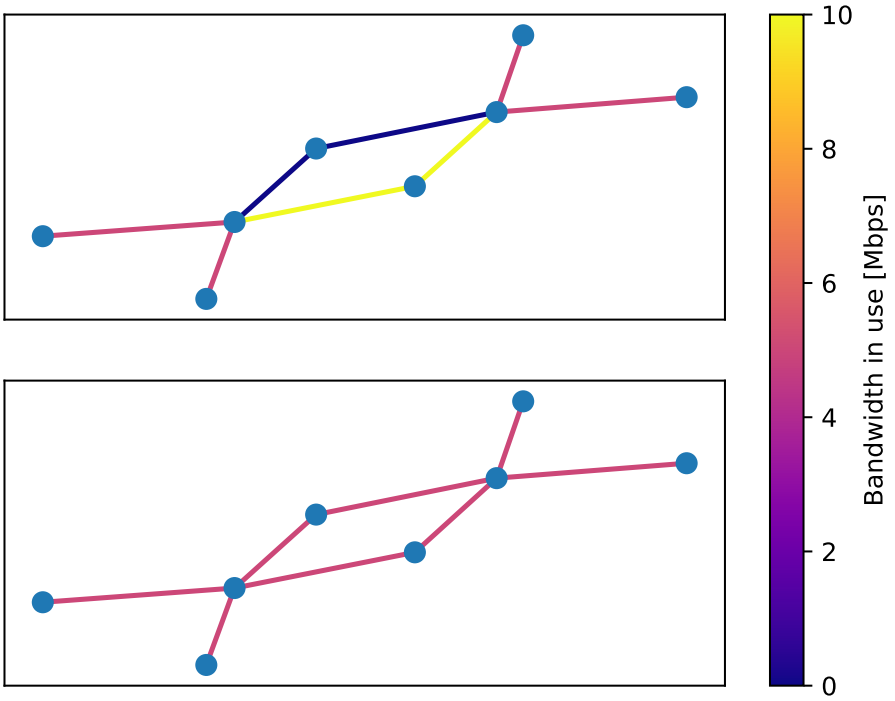

Figure 7.3: Lower bandwidth requirements for interconnecting links. As each flow can be routed differently, a single link no longer needs to have enough bandwidth to transmit all the traffic.

`n1.b.IPCP`, who is the Raft leader for the current term, to IPCP `n1.c.IPCP`. In the case of leader failure, the replicas have all the necessary information to be able to take over and serve client requests.

7.1 Future work

It has been agreed to focus on a basic, simple scenario which means there are several issues which could use improvement and could form the basis of a future work. These have all been thoroughly considered and discussed; some of them have been even partially implemented but were later removed from the codebase due to being deemed lacking.

7.1.1 Flow reshuffling

In its current state, the algorithm does not react to changes in the network graph. This means, among other things, that if the node through which a reservation flows fails, the flow effectively dies, due to having static routes. On first look, the solution is quite simple. The Raft leader needs to be notified whenever a link goes down to try find an alternate path (and update the replicated state as well as static routes to contain the new path) or to inform the end IPCPs that the bandwidth can no longer be guaranteed and the flow should therefore be terminated.

One of the problems with this is that on each link state change, the leader would need to go through *every active flow* and iterate over *the whole path* to find paths that are no longer

```
IPCP name: n1.b.IPCP

Lower Flow Database:
    Local: n1.d.IPCP, Remote: n1.c.IPCP, Total: 10Mbps, Free: 10Mbps
    Local: n1.b.IPCP, Remote: n1.e.IPCP, Total: 10Mbps, Free: 10Mbps
    Local: n1.b.IPCP, Remote: n1.c.IPCP, Total: 10Mbps, Free: 0Mbps
    ...

Flow reservation table:
     n1.a.IPCPn1.c.IPCP4: n1.a.IPCP->n1.c.IPCP
        (n1.a.IPCP,n1.b.IPCP,n1.c.IPCP,) : 5Mbps
     n1.a.IPCPn1.c.IPCP3: n1.a.IPCP->n1.c.IPCP
        (n1.a.IPCP,n1.b.IPCP,n1.c.IPCP,) : 5Mbps

IPCP name: n1.c.IPCP

Lower Flow Database:
    Local: n1.d.IPCP, Remote: n1.c.IPCP, Total: 10Mbps, Free: 10Mbps
    Local: n1.c.IPCP, Remote: n1.d.IPCP, Total: 10Mbps, Free: 10Mbps
    Local: n1.c.IPCP, Remote: n1.b.IPCP, Total: 10Mbps, Free: 0Mbps
    ...

Flow reservation table:
     n1.a.IPCPn1.c.IPCP4: n1.a.IPCP->n1.c.IPCP
        (n1.a.IPCP,n1.b.IPCP,n1.c.IPCP,) : 5Mbps
     n1.a.IPCPn1.c.IPCP3: n1.a.IPCP->n1.c.IPCP
        (n1.a.IPCP,n1.b.IPCP,n1.c.IPCP,) : 5Mbps
```

Figure 7.4: Replicated state between nodes in the Raft cluster.

valid and need to be recomputed. This is significantly more computationally intensive than the initial reservation or removing such reservation, and in the case of DIFs with many flows and long paths, especially with a link flapping, could significantly impact the leader's performance.

In addition, this problem is actually more general than it seems. It might be helpful to recompute the paths of some flows that are valid, and route them differently, to free up paths for the flows that have lost their reserved path. The opposite scenario, a new IPCP enrolling in a DIF might also be a good opportunity to shuffle the flows around, as new, more effective (shorter) paths might have appeared. In fact, even a new flow being requested, or a flow being deallocated, could benefit from such flow reshuffling. All of these approaches would potentially allow better overall bandwidth utilization.

The downsides of moving flows around are, besides the aforementioned increasing computational complexity, disruptions to the traffic flow and the overall stability of the system. The problem becomes, from a simple decision problem (does an available path exist), an optimization problem which tries to distribute the flows in the most optimal way. It is a series of trade-offs, which suggests it might be good to implement these behaviours as toggleable policy (parameters), and leave deciding which of them and with what parameters will be useful to the network administrators. However, restricting the dynamic behaviour to only link failures, and only trying to find a new path (no flow shuffling being done) would be a reasonable first step that would improve the system's reliability.

7.1.2 Support for unlimited flows

Currently, all unlimited flows (those that do not request a particular bandwidth) are either rejected or forced a bandwidth value depending on the policy parameters. This prevents any unwanted disruptions from entering into the system. However, this means that every flow gets a reservation taking up bandwidth, possibly preventing other, limited flows, from being accepted.

A more sophisticated solution would be to dynamically rate limit the free flows to be able to only use the unreserved bandwidth. This would require the Raft leader to distribute the current limit to each unlimited flow whenever a flow gets (de)allocated.

One approach might be to calculate the average free bandwidth for each link, such as:

$$BwAvg = \frac{BwTotal - BwRes}{NFlows}$$

where $BwTotal$ is the maximum bandwidth of the link, $BwRes$ is the bandwidth that is currently reserved for this link, and $NFlows$ is the number of unlimited flows flowing through the link. An unlimited link would then get assigned the minimum (bottleneck) $BwAvg$ of all the links it goes through. This solution is not optimal, and potentially leaves some bandwidth unused; it is however fairly simple to calculate.

Another approach would be to set not divide the bandwidth among the flows and use the minimum $BwFree = BwTotal - BwRes$ along the path as the limit. Congestion control mechanisms would come into play to divide the bandwidth according to the flow's demands. However, as the overall rate limit will be higher than the capacity of the interface, this might negatively impact the flows with reservations.

No matter the calculation model, it is necessary to update the structure of the flow's EFCP instance in the kernel with the value received from the Raft leader. No such interface currently exists.

7.1.3 Flow splitting

Flow splitting, using multipath routing, is reminiscent of the concept of augumenting paths from the Ford-Fulkerson maximum flow algorithm. Instead of using a single path that has the necessary capacity, the flow is split among several different paths so that its bandwidth requirements can be met.

Aside from having to tell each route how to divide the flow, there are some additional issues this approach presents. As each path might have different characteristics (number of hops, latency), issues such as packet reordering and head-of-line blocking could occur. It makes the flow management more complex in general, and in combination with some of the more dynamic approaches described above would mean even more information to keep track of.

Chapter 8

Conclusion

In the first part of this thesis, we have looked at congestion in networks from a theoretical point of view. We have studied the causes of congestion and how different parameters affect its behaviour, such as queue length. We looked at two possible approaches of avoiding congestion — proactive and reactive ones.

A proactive approach, like the one implemented in this thesis, attempts to prevent congestion by reserving some of the available bandwidth for each flow. As the entity in control of flow allocation will never allow the reserved bandwidth to exceed the total available bandwidth, and no flow is able to use more than they have been assigned, it prevents the cause of congestion, and therefore congestion itself, from occurring. While this approach is generally simpler to implement and model, the bandwidth utilization efficiency suffers due to very static nature of the solution. Any reserved bandwidth not utilized by the flow it belongs to cannot be used by another flow.

A reactive approach, represented by the various congestion control algorithms included as part of TCP, attempts to prevent congestion by dynamically adjusting the sending rate of all the active flows based on some feedback from the network. We have learned that there exists an optimal operating point called the *knee*, and that all congestion control algorithms try to match the overall sending rate to be as close to the knee as possible. While this allows the bandwidth to be better distributed in theory, as any residual bandwidth will be claimed by some flow that needs it, such an approach is reliant on the quality of the feedback it gets from the network, and any assumptions it might make about it and the current state.

Following general congestion control theory, we have studied some examples of current, real-world algorithms implemented and used in TCP. TCP Tahoe was picked as the reference example of a simple congestion control algorithm due to its historical significance. This protocol was studied in detail to see how it maps to the theory learned and what general mechanisms it contains that might be tweaked by newer algorithms.

As examples of more modern, state of the art algorithms were picked TCP CUBIC, QUIC and TCP BBR. TCP CUBIC has been the default congestion control algorithm for the Linux kernel for some time, and is therefore the baseline to which other algorithms are compared, performance-wise. QUIC and TCP BBR are both efforts from Google to create a better algorithm that fits the needs of the modern web. Each was analyzed to see how it operates and what core ideas and new approaches it brings to solve the problem of congestion in computer networks. It has been found that these newer algorithms are better suited for the environment of current Internet, and achieve better results especially in the case of long fat pipes (paths with a long bandwidth delay product) where an older algorithm, such as Tahoe, does not grow the congestion window fast enough. However,

both QUIC and BBR, although being newer and meant to be replacements for CUBIC, have specific scenarios where they perform worse. In addition, all of these algorithms suffer from fairness issues, either intra-algorithm for e.g. various flow RTT classes, or between the algorithms themselves.

A proactive bandwidth reservation scheme for flow allocation has been proposed and designed for RINA, and implemented in rlite. This scheme, implemented as a flow allocator policy, uses the *average bandwidth*[1] value of a flow request to try find a suitable path in the DIF graph. If no viable path is found, the request is denied, else this bandwidth is reserved for the lifetime of the flow. This policy works as logically centralized, but is in fact distributed to provide fault tolerance within the DIF. Using the Raft distributed consensus algorithm, the reservation information is replicated onto several nodes, each being able to take over to serve client requests in case of node failure.

Experiments have been done to show how this reservation scheme works with a higher number of flows, and how it allows for quite evenly distributed utilization of bandwidth. Scenarios have been described, where the ability to route each flow differently allows the backbone/core links to have lower bandwidth requirements.

Finally, a section is devoted to discussing future improvements to this flow allocation scheme. As most of them contain some tradeoffs to be made, it is advised that if these should be implemented, they should be a configurable part of the policy. The first extension is can be described as flow reshuffling — there are several events where trying to reroute a flow might allow for better overal utilization. The second deals with flows which do not include the average bandwidth as part of the flow allocation request; in the current state, these are either rejected for forcibly rate limited. Two possible approaches are presented how to allow these flows to use the available unreserved bandwidth in a more dynamic way. Lastly, the idea of splitting the flow among several paths, akin to multipath TCP, to increase bandwidth utilization is briefly considered.

[1] The Error and Flow Control Protocol uses this value to rate limit the flow.

Bibliography

[1] Braden, R.: Requirements for Internet Hosts - Communication Layers. STD 3. RFC Editor. October 1989.
Retrieved from: http://www.rfc-editor.org/rfc/rfc1122.txt

[2] Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation*. 2006. page 335–350.

[3] Cardwell, N.; Cheng, Y.; Gunn, C. S.; et al.: BBR: Congestion-Based Congestion Control. *Queue*. vol. 14, no. 5. October 2016: pp. 50:20–50:53. ISSN 1542-7730. doi:10.1145/3012426.3022184.

[4] Cardwell, N.; Cheng, Y.; Yeganeh, S.; et al.: BBR Congestion Control. Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00. IETF Secretariat. July 2017.
Retrieved from: http://www.ietf.org/internet-drafts/draft-cardwell-iccrg-bbr-congestion-control-00.txt

[5] Cheng, Y.; Cardwell, N.; Yeganeh, S.; et al.: Delivery Rate Estimation. Internet-Draft draft-cheng-iccrg-delivery-rate-estimation-00. IETF Secretariat. July 2017.
Retrieved from: http://www.ietf.org/internet-drafts/draft-cheng-iccrg-delivery-rate-estimation-00.txt

[6] Day, J. D.: *Patterns In Network Architecture: A Return to Fundamentals*. Prentice Hall. 2010. ISBN 0137063385.

[7] Fletcher, J. G.; Watson, R. W.: Mechanisms for a Reliable Timer-Based Protocol. *Computer Networks*. vol. 2. 1978: pp. 271–290.

[8] Gancarz, M.: *Linux and Unix Philosophy*. Digital Press. 2003. ISBN 1555582737.

[9] Gross, J. L.; Yellen, J.; Zhang, P.: *Handbook of Graph Theory (Discrete Mathematics and Its Applications)*. CRC Press. 2014. ISBN 978-1-4398-8018-0.

[10] Ha, S.; Rhee, I.; Xu, L.: CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.*. vol. 42, no. 5. July 2008: pp. 64–74. ISSN 0163-5980. doi:10.1145/1400097.1400105.

[11] Hamilton, R.; Iyengar, J.; Swett, I.; et al.: QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2. Internet-Draft draft-hamilton-early-deployment-quic-00. IETF Secretariat. July 2016.
Retrieved from: http://www.ietf.org/internet-drafts/draft-hamilton-early-deployment-quic-00.txt

[12] Hock, M.; Bless, R.; Zitterbart, M.: Experimental evaluation of BBR congestion control. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE. 2017. pp. 1–10.

[13] Huston, G.: *ISP Column - January 2019* [online]. 2019-01 [Accessed 2019-01-15]. Retrieved from: https://www.potaroo.net/ispcol/2019-01/bgp2018-part1.html

[14] IETF HTTP Working Group: *HTTP/2* [online]. 2018-10-09 [Accessed 2019-01-15]. Retrieved from: https://http2.github.io/

[15] Jacobson, V.: Congestion Avoidance and Control. In *Symposium Proceedings on Communications Architectures and Protocols*. SIGCOMM '88. New York, NY, USA: ACM. 1988. ISBN 0-89791-279-9. pp. 314–329. doi:10.1145/52324.52356.

[16] Kakhki, A. M.; Jero, S.; Choffnes, D.; et al.: Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols. In *Proceedings of the 2017 Internet Measurement Conference*. IMC '17. New York, NY, USA: ACM. 2017. ISBN 978-1-4503-5118-8. pp. 290–303. doi:10.1145/3131365.3131368.

[17] Keshav, S.: *Congestion Control in Computer Networks*. PhD. Thesis. University of California at Berkeley. 1991.

[18] Koutenský, M.: rlite: building scalable networks the recursive way. In *Excel@FIT*. 2019.

[19] Lamport, L.: The Part-Time Parliament. In *ACM Transactions on Computer Systems 16, 2*. 1998. pp. 133–169.

[20] Larry L. Peterson, B. S. D.: *Computer Networks: A Systems Approach*. The Morgan Kaufmann Series in Networking. Morgan Kaufmann. third edition. 2003. ISBN 1-55860-832-X.

[21] Maffione, V.; Koutenský, M.: *rlite: A light RINA implementation* [online]. 2019-01-10 [Accessed 2019-01-15]. Retrieved from: https://github.com/rlite/rlite

[22] Ongaro, D.: *Consensus: Bridging Theory and Practice*. PhD. Thesis. Stanford University. 2014.

[23] Ongaro, D.; Ousterhout, J. K.: In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*. 2014. pp. 305–319.

[24] Park, T.; Lee, J.; Kim, B.: Design and Analysis of High Performance TCP. In *High Performance Computing and Communications*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2006. ISBN 978-3-540-39372-6. pp. 370–379.

[25] Postel, J.: Transmission Control Protocol. STD 7. RFC Editor. September 1981. Retrieved from: http://www.rfc-editor.org/rfc/rfc793.txt

[26] Pouzin Society: *Terminology | Pouzin society* [online]. Uknown [Accessed 2019-01-15]. Retrieved from: http://pouzinsociety.org/education/terminology

[27] Roskind, J.: *QUIC: Design Document and Specification Rationale* [online]. 2015-06-05 [Accessed 2019-01-15].
Retrieved from: https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit

[28] Scholz, D.; Jäger, B.; Schwaighofer, L.; et al.: Towards a Deeper Understanding of TCP BBR Congestion Control. 2018.

[29] The Chromium Project: *QUIC, a multiplexed stream transport over UDP* [online]. Uknown [Accessed: 2019-01-15].
Retrieved from: https://www.chromium.org/quic

[30] The Chromium Project: *SPDY: An experimental protocol for a faster web* [online]. Unknown [Accessed 2019-01-15].
Retrieved from: https://www.chromium.org/spdy/spdy-whitepaper

[31] The Linux man-pages project: *ioctl(2) Linux Programmer's Manual*. Fifth edition. 05 2017.

[32] Veselý, V.: *A New Dawn of Naming, Addressing and Routing on the Internet*. PhD. Thesis. Brno University of Technology. 2015.

[33] Wang, R.; Pau, G.; Yamada, K.; et al.: TCP startup performance in large bandwidth networks. In *IEEE INFOCOM 2004*, vol. 2. March 2004. ISSN 0743-166X. pp. 796–805 vol.2. doi:10.1109/INFCOM.2004.1356968.

[34] Welzl, M.: *Network Congestion Control: Managing Internet Traffic*. John Wiley & Sons Ltd. 2007. ISBN 0-470-02528-X.

[35] Xu, L.; Harfoush, K.; Rhee, I.: Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM 2004*, vol. 4. March 2004. ISSN 0743-166X. pp. 2514–2524 vol.4. doi:10.1109/INFCOM.2004.1354672.

Appendix A

Contents of the CD

The included CD contains these directories and files:

- The electronic version of this thesis, called `xkoute04.pdf`
- A directory called `tex` containing the LaTeX source files
- A directory called `src` containing the source code for rlite

Appendix B

List of abbreviations

- **ACK** — ACKnowledgement
- **AE** — Application Entity
- **AP** — Application Process
- **APN** — Application Process Name
- **ARPANET** — Advanced Research Projects Agency Network
- **BDP** — Bandwidth-Delay Product
- **BFS** — Breadth First Search
- **BGP** — Border Gateway Protocol
- **CDAP** — Common Distributed Application Protocol
- **CEP** — Connection EndPoint
- **DAF** — Distributed Application Facility
- **DIF** — Distributed IPC Facility
- **DMS** — DIF Management System
- **DTCP** — Data Transfer Control Protocol
- **DTP** — Data Transfer Protocol
- **EFCP** — Error and Flow Control Protocol
- **GNU** — GNU is Not Unix
- **HTTP** — HyperText Transfer Protocol
- **I/O** — Input/Output
- **IP** — Internet Protocol
- **IPC** — InterProcess Communication

- **IPCP** — IPC Process
- **LFDB** — Lower Flow DataBase
- **LGPL** — GNU Lesser General Public License
- **MPL** — Maximum Packet Lifetime
- **NCP** — Network Control Program
- **OSI** — Open Systems Interconnection
- **PM** — Protocol (state) Machine
- **RFC** — Request For Comments
- **RIB** — Resource Information Base (RINA), alternatively Routing Information Base (BGP)
- **RINA** — Recursive InterNetwork Architecture
- **RTT** — Round-Trip Time
- **SDN** — Software Defined Networking
- **TCP** — Transmission Control Protocol
- **TLS** — Transport Layer Security
- **UDP** — User Datagram Protocol
- **UIPCP** — Userspace IPCP (daemon)