Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Robin Berner

Improving Performance with Network-aware Scheduling Algorithms

Semester Thesis SA-2019-24
October 2019 to January 2020

Tutor: Prof. Dr. L. Vanbever
Supervisor: Albert Gran Alcoz
Co-Supervisor: Alexander Dietmüller

Abstract

This thesis aims at exploring the possibilities of a novel algorithm to control congestion and increase the performance of the network by influencing the packet scheduling of a switch with the help of information gathered in the network. Contrary to that, in today's internet, congestion is predominantly controlled by relaying information back to the host with little to no help from the network. Not only does this new approach ensure fairness between the host by excluding them from the process, it also reduces the reaction time to congestion by lowering the number of involved elements.
I propose the theoretical concept of using upstream information to inform switches along the path about upcoming bottlenecks and show how to use this information to free up bandwidth in the network, that was previously used by traffic which has a high chance of being dropped along the path.
Using the programmable data plane of P4 to implement the control logic as well as keeping track of the state of the network in the switch itself, I develop my proposed design of the algorithm to prove its functionality and to show its usefulness.

Contents

Chapter 1

Introduction

Congestion control is a large and actively researched topic, which aims at improving the overall performance of a network by resolving and preventing congestion, be it in a datacenter or a world-spanning network like the internet. Congestion on a switch means packets arrive quicker than they get processed and sent back out, leading to the buffer filling up and packets getting dropped once the buffer reaches its limit. If congestion is left unattended it can lead to a significant decrease in performance up to a congestive collapse of the network (e.g. collapse of the internet in 1986).
This project seeks to use information about the routing of a packet on future hops to influence the routing at an earlier stage, lowering the chance of new traffic experiencing congestion and overall utilize the existing bandwidth better. This can be achieved by changing the packet scheduling on individual switches in the network.

1.1 Motivation

Scheduling algorithms solve a variety of issues and have been researched intensively. Some prominent examples from recent years include pFarbic [1], which minimizes flow completion times, or Approximating Fair Queueing [4], providing a fair share across flows. However none of them use upstream information, meaning information acquired on switches further along the path. By using upstream information, this thesis seeks a new possibility to optimize the local scheduling in order to increase the overall performance of the network.
Additionally many congestion control algorithms (e.g. TCP congestion control) involve the traffic generating hosts in order to detect and resolve congestion. Not only does this mean hosts in a network need to be trusted and cooperate in order to achieve fairness, it means the reaction time is longer as well. Detecting congestion directly in the data plane enables the acquisition of the needed information directly inside the network and does not rely on information sent by a host. This ensures fairness by excluding the hosts from the process and shortens the response time.

1.2 The Task

During the course of this thesis multiple tasks need to be solved and answered, starting with the design of the protocol from scratch. The protocol needs to have a solution to detect congestion on every switch inside the network individually. Furthermore it needs to be able to inform individual switches about the state of the congestion on other switches, specifically switches further along the routing path of flows passing through. Finally the informed switches have to change their packet scheduling to improve the overall network performance by reducing the amount of congested traffic inside it.
Once a protocol is defined, different approaches and algorithms solving individual aspects need to be explored to satisfy varying requirements and limitations inside a network, such as round-trip times and remaining computational resources on a switch, to ensure a certain bandwidth. At last an implementation of the proposed protocol needs to verify its functionality.

1.3 Related Work

Detecting congestion in a network is not a simple task, many different approaches already exist today and it is hard to say if there is an optimal solution. While certain papers [5] determine the state of congestion according to the number of consecutive packets that exceed certain queueing and processing delays, others [3] show that bufferbloat has similar symptoms and is often mistaken as congestion. However the advantages of detecting congestion in this fashion lie in its simplicity and relatively low complexity which enables it to easily run at line rate.
Existing scheduling algorithms such as pFarbic and Fair Queueing also encourage using the network itself to support congestion control with their results, while [2] encourages to keep searching for alternatives, since there does not exist a single universal packet scheduling algorithm able to replicate all viable schedules.

1.4 Overview

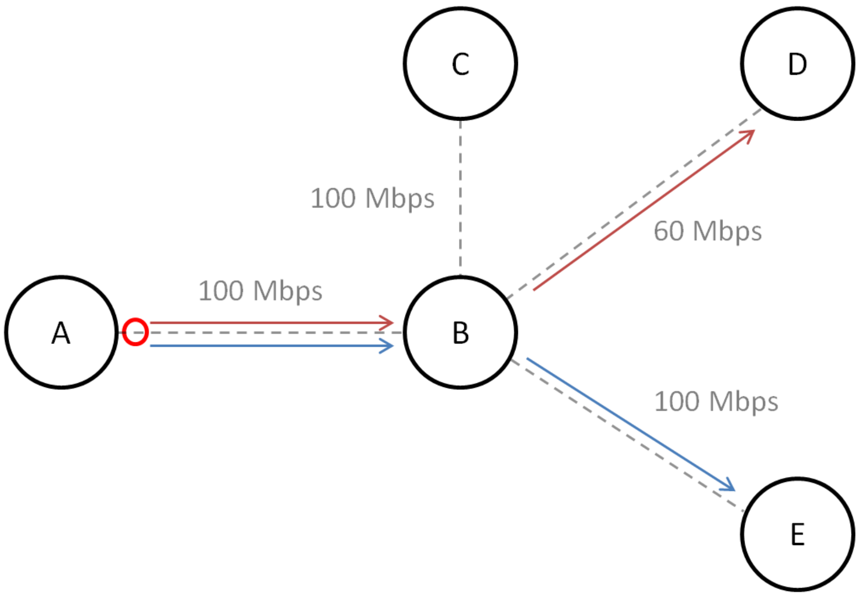
Chapter 2 specifies the exact problem we want to solve in this thesis, illustrates it with an example and shows what use solving it has.
Chapter 3 presents the proposed protocol to solve the problem described in Chapter 2 and discusses how to implement the protocol in P4.
Chapter 4 evaluates the design presented in Chapter 3.
Chapter 5 states what is missing and how the project can be improved further.
Chapter 6 quickly sums up the whole thesis and what it achieved.

Chapter 2

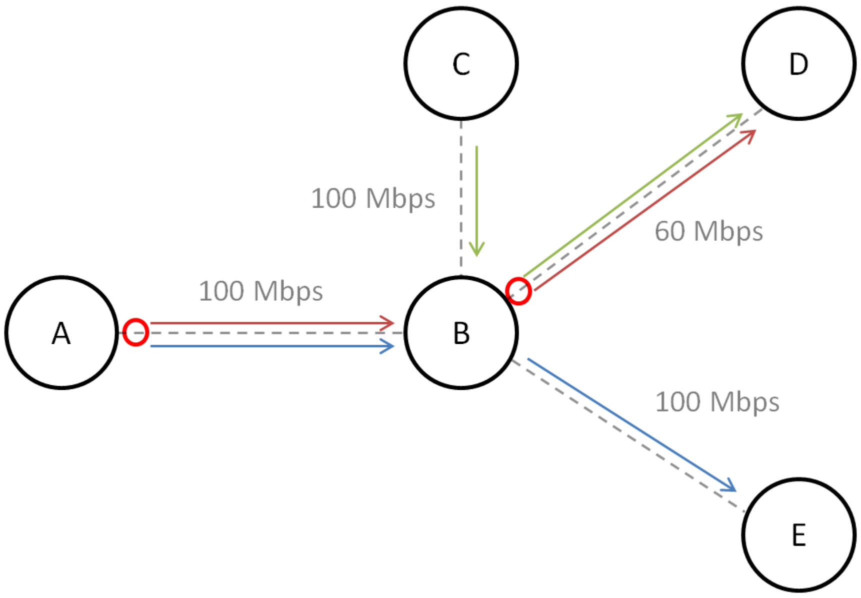
The Problem

This thesis addresses the issue of using upstream information inside the network, in order to react to congestion earlier and free wasted bandwidth. When congestion occurs inside a network, traffic gets dropped before reaching its destination. Ideally knowing that the traffic will get dropped, one would avoid routing this traffic at all, since it occupies bandwidth that could be used for traffic which reaches its target and therefore increasing the effective throughput. If switches inside a network are able to send information upstream, back to where the packet came from, previous switches can adjust their packet scheduling in order to lower the chance of routing to-be-dropped traffic.
In today's internet most switches are not programmable and simply forward the traffic according to their preconfigured tables. Trying to send information inside the network is not possible directly in the data plane and would involve a local controller to analyze incoming traffic, significantly increasing the delay of the detection of the problem. A new approach such as programmable data planes needs to be used in order to resolve this issue.

The best way to show the problem and the improve in performance of solving it is an example. Consider Figure 2.1a as the initial situation in the network. The red and blue flow each transmit traffic at a rate higher than 50 Mbps via a protocol like UDP with no built in congestion control. This results in congestion on the link between switch A and switch B, where each flow achieves an average throughput of 50 Mbps, adding up to a total of 100 Mbps used, which corresponds to the link capacity of the congested link. Switch B does not have any congestion on its egress ports as the link capacity is higher than 50 Mbps in both cases.
Now consider Figure 2.1b, where the additional flow in green is present, transmitting traffic at a rate higher than 100 Mbps via a protocol like UDP as well. The situation at switch B changes and it acts according to the state of each individual port.
The egress port to switch D receives 50 Mbps of the red flow and 100 Mbps of the green flow, resulting in a total of 150 Mbps which needs to be forwarded through a link of 60 Mbps. The available bandwidth gets split up between the two flows, each flow getting a share of the total 150 Mbps. On average it can be calculated as the percentile of the flows traffic compared to the total traffic, meaning the red flow gets $\frac{50}{150} = \frac{1}{3}$ or 20 Mbps of the link, whereas the green flow gets $\frac{100}{150} = \frac{2}{3}$ or 40 Mbps of the link on average.
However, the egress port to switch E is not congested in any way, the 50 Mbps traffic of the blue flow can be forwarded without dropping any packets.
All of this adds up to a total of 110 Mbps traffic that reaches its destination. This leaves room for optimization since the blue flow could achieve a higher throughput than 50 Mbps, up to 100 Mbps, while the red and green flow combined still preserve their combined throughput of 60 Mbps, resulting in a better performance of the network overall. This increase in performance can be achieved by adjusting the scheduling on switch A such that the blue flow gets prioritized over the red flow, since it has a higher chance of successfully reaching its destination.



(a) One congested link, marked with a red circle



(b) Two congested links, marked with a red circle

Figure 2.1: An example scenario showing the problem

Chapter 3

Design

In this chapter, the protocol to solve the problem described in Chapter 2 is presented and an implementation of its core functionality discussed. Section 3.1 shows the theory of the protocol and lists different approaches to achieve the given functionality. Section 3.2 describes the implemented code which was used to test and verify the proposed protocol.

3.1 Protocol

The protocol consist of four steps. First, congestion has to be detected (Subsection 3.1.1). As soon as congestion is detected on a switch, affected flows have to be marked to indicate the congested state to following switches (Subsection 3.1.2). If marked flows are part of another congestion, a reduction in priority needs to be applied by modifying the packet scheduling (Subsection 3.1.3). Finally the state needs to be reset to normal in order to not penalize future flows at the end of a congestion (Subsection 3.1.4).

3.1.1 Congestion Detection

As we have seen in Chapter 2, the protocol needs to be able to detect congestion on individual links. Congestion detection has to happen on every egress port of a switch separately and for each flow individually to ensure that only congested flows have lower priority and not every flow across the switch is penalized.
Congestion detection can be done in many different ways, each with its own advantages and drawbacks, three different approaches will now be presented and discussed.

One method of determining the state of congestion uses the information about the status of a buffer of an egress port. If a link is congested, packets queue up faster than they get dequeued, which in return means the buffer is filling up and once it reaches its limit, packet drop will occur. Therefore it is possible to determine the state of congestion by reading out the occupation of a buffer and checking if it is full over a certain amount of time or packets. This method does not need a lot of logic to implement and runs fairly quick, however setting threshold values for the amount of packets or time needed, as well as what is considered a full buffer is no easy task and can not be answered universally. It is also possible that false detection occurs due to the bursty behaviour of the internet or bufferbloat. Bufferbloat means that a certain occupation of the buffer persist over a longer period of time even though the same amount of traffic that comes in also leaves the buffer. If bufferbloat occurs at a full buffer, it looks like congestion but no traffic is dropped.

Another method separates the time into intervals, so called buckets. Each bucket is only valid for one time interval T, during which all packets of a flow get counted into the same bucket. After a time interval T passed, we close the bucket and open a new one to restart counting. Congestion can now simply be determined by looking at the last X buckets and evaluating if enough of them counted over a threshold.
Again, determining the threshold values is not trivial and the logic needed is more complicated

than checking the buffer status. Higher memory usage for keeping the state of the buckets over a longer period of time is also a major drawback, but in return the expected behaviour is more stable than only judging based upon the status of the buffer.

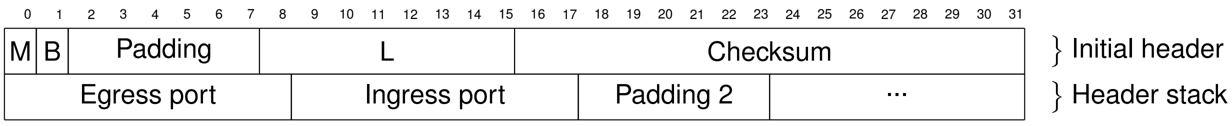
Last but not least, a fairly similar method to the aforementioned method using buckets aims at reducing the memory overhead. Instead of keeping several buckets alive, one could simply low-pass filter the counter after a time interval T has passed by taking

$$\begin{aligned} old\ bucket\ value &= \frac{old\ bucket\ value}{2} + \frac{new\ bucket\ value}{2} \\ new\ bucket\ value &= 0 \end{aligned}$$

and start counting from zero again in the new bucket. Congestion is now only based on the value of the old bucket instead of evaluating multiple bucket. This reduces the amount of memory, since only 2 buckets need to be alive at any given time per flow, but in return reduces the precision.

3.1.2 Marking

If congestion for a flow is detected, the switch sets its state to mark the given flow from now on. The flow needs to be marked in order to inform upcoming switches to relay information to the marking switch. This is done by adding a new header to the packet, consisting of an initial header for general information and a header stack with information about every hop taken from the marking switch. The header follows the given structure.



M	Signalizes if a flow is marked or not. If it is set to 1, the flow is marked. If it is set to 0, the flow is not marked
B	Signalizes if the packet is building the header stack or removing elements from it. If it is set to 1, the header stack is being built and each hop adds a new element to the header stack. If it is set to 0, the packet is routed according to the information in the header stack and each hop pops its own element from the stack
Padding	Align the header to 8 bits
L	Counts the amount of elements in the header stack
Checksum	Used to verify the integrity of the header
Egress Port	Egress port taken on the hop which added the element to the header stack
Ingress Port	Ingress port taken on the hop which added the element to the header stack
Padding 2	Align the header stack element to 8 bits

After applying normal longest prefix match according to the IPv4 header, each switch in the network now needs to differentiate following cases and additionally apply the corresponding steps.
If the flow is not marked, meaning either the bit M is 0, or the header is not attached, two cases need to be considered. If congestion for this flow is detected on the switch, it will start marking each packet of the flow with the initial 32 bit long header and the first element of the header stack. If no congestion is detected the switch does nothing.
If the flow is marked and the B bit is set to 1, there are two different cases again. If congestion is detected on this switch as well, send a feedback packet back by creating a copy and set the bit B to 0, the path is defined by the header stack. If no congestion was detected, the switch will add another element to the header stack with the information about this hop and increase the L field in the initial header by one.

If the flow is marked and the B bit is set to 0, the received packet is a feedback packet. IPv4 routing needs to be ignored and instead the information of the header stack is used, additionally remove this hop from the header stack and decrease the L field in the initial header by one.

3.1.3 Reaction

Receiving a feedback packet with B set to 0 and L set to 1 signalizes a switch that it initially started marking this flow. Knowing this flow is involved in another congestion later on its path the switch reduces its priority. Every packet of this flow will now have a reduced priority applied to it.

3.1.4 Resolve Congestion

By changing the priority of a flow in response to detecting congestion, the behaviour of the network changes and previously congested links may no longer be congested. However simply reverting the actions taken as soon as no congestion is seen anymore could lead to an oscillating behaviour if done too quickly. Therefore a minimum time span needs to be defined for which a state should be kept. However, finding a global value may not be feasible depending on the network structure. The round-trip time in between the switches detecting congestion should be considered as well, in order to not reset the state before any traffic with different scheduling applied to it arrives at the second detecting switch.

3.2 Implementation

The protocol defined in 3.1 has been coded in P4 and can be found in the GitLab [1]. In this section the individual elements needed in the processing pipeline and what tasks they need to fulfill are briefly discussed. Since section 3.1 already defined the protocol, the details on how it works can be studied there.
There are 3 main elements needed to realize the code, the ingress and egress of the processing pipeline in a switch are done in P4, while the local controllers of the switches are done in Python.
The congestion detection needs to happen in the egress as only there we have access to standard_metadata elements like the dequeue queue depth and know the correct time at which packets get dequeued. In the provided code the congestion detection is done using the buffer status mentioned in 3.1.1. Once the counter of consecutive packets experiencing a high buffer occupation reaches the threshold, a copy of the packet is recirculated to the ingress. This packet will be used to inform the ingress about the congestion, since the egress and the ingress share no memory in P4. The counters are implemented using a Count-min sketch in order to scale to a large amount of flows, since each flow needs its own counter.
The ingress takes care of multiple things. All packets are still routed according to standard IPv4 longest prefix match routing. Feedback packets which need to be routed according to the header stack overwrite the IPv4 routing at a later stage. The marking as defined in subsection 3.1.2 needs to be applied and the recirculated packets from the egress need to be processed in order to set the state to mark the flow. Additionally feedback packets need to be processed according to subsection 3.1.3. The state for marked flows and for congested flows is kept by using Bloom filters to ensure scalability. Lastly the scheduling of the packets needs to be adjusted to lower the priority of congested flows. This is done by hashing the packet and checking the state of the Bloom filter, which determines the priority of the flow to which the packet belongs.
The controller is involved to manage the time. Using a timeout to reset the state of congested flows on a switch, the controller not only needs to keep track of the time, but also needs to reset the Bloom filters keeping the state. Additionally when using certain congestion detection algorithms, such as using buckets to count, the controller needs to signalize the switch when a time interval has passed.
Finally the controller also checks if a link leaves the network and if so, the IPv4 table in the ingress also marks the packet's metadata in order to remove the custom header and header

[1] https://gitlab.ethz.ch/nsg/student-projects/sa-2019-24_improving_performance_with_network-aware_scheduling_algorithms

stack in the egress once the packet is done processing. This makes it possible to run applications in the network that are not aware of the additional headers of the protocol.

Chapter 4

Results

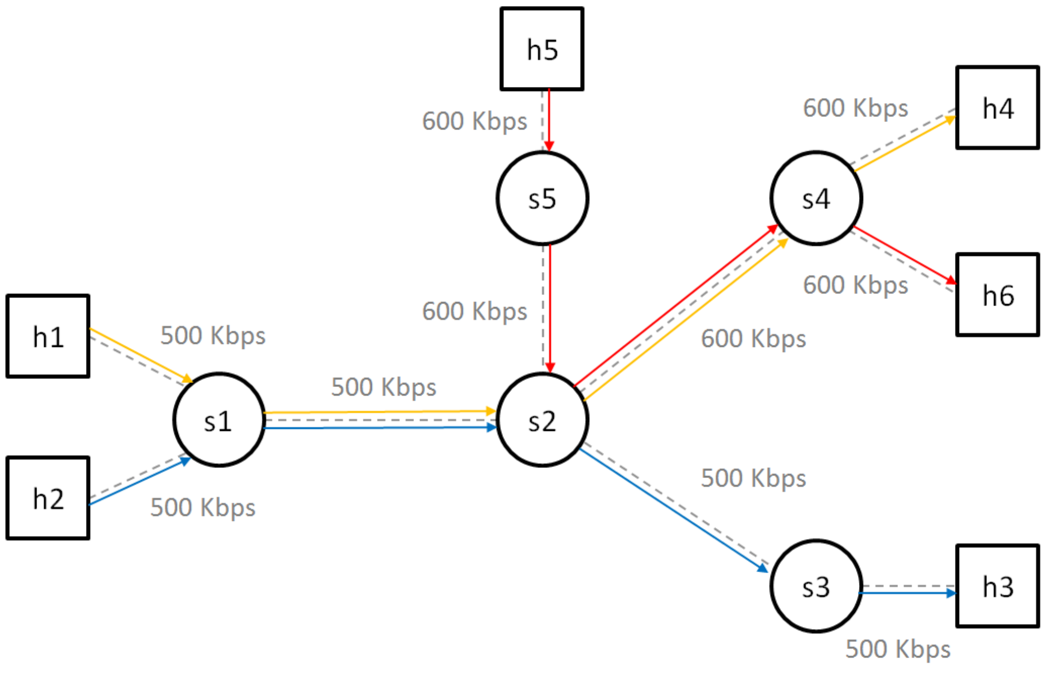


Figure 4.1: Network overview for evaluation

To verify the functionality of the designed protocol in Chapter 3, the implementation described in 3.2 was tested on the setup in Figure 4.1. The three flows are simulated by using the tool Iperf [1]. Hosts h1, h2 and h5 are set up as clients sending UDP traffic by using

```
$ iperf –c <target IP> –b 1000000 –t 120 –i 1
```

The target IP can easily be found in the output of the mininet setup, while the bandwidth given with the flag -b needs to be larger or equal to the bandwidth on the links connecting the host to the switch. Hosts h4, h6 and h3 which are set up as receiving servers and measure the bandwidth received in one second intervals by using the command

```
$ iperf –s –u –i 1
```

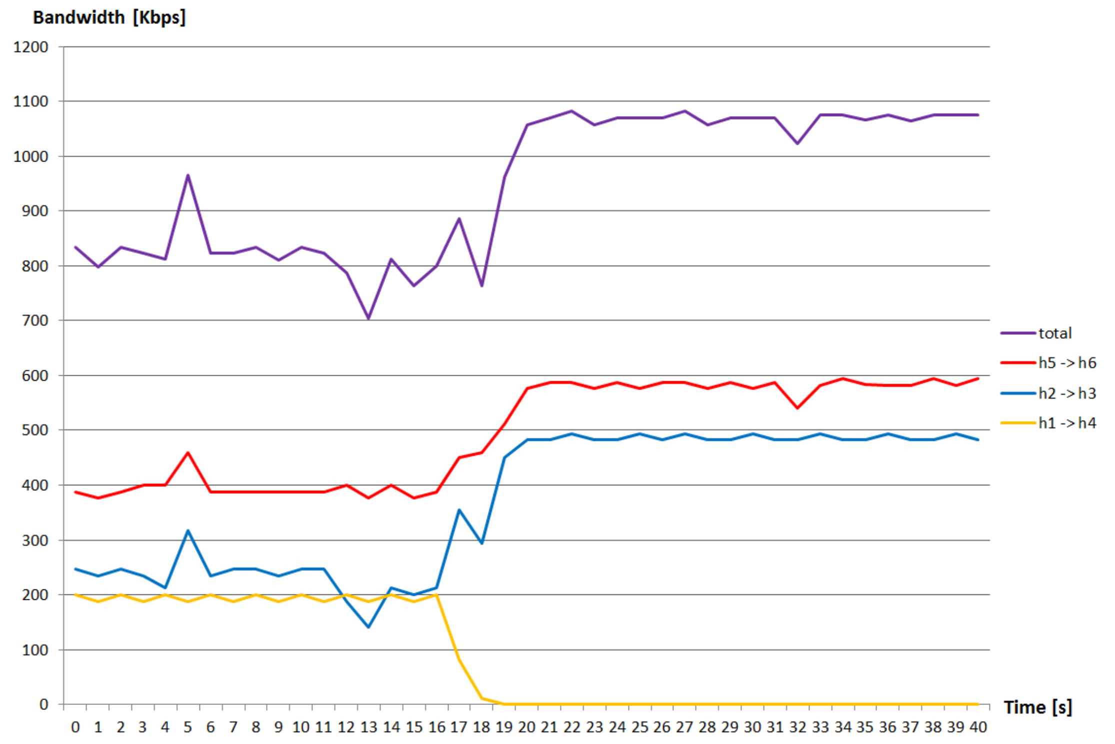
At the start of the test without any interference of the protocol the bandwidth is expected to be split up between all three flows. The blue flow from h2 to h3 has to share the link between

[1] https://iperf.fr/

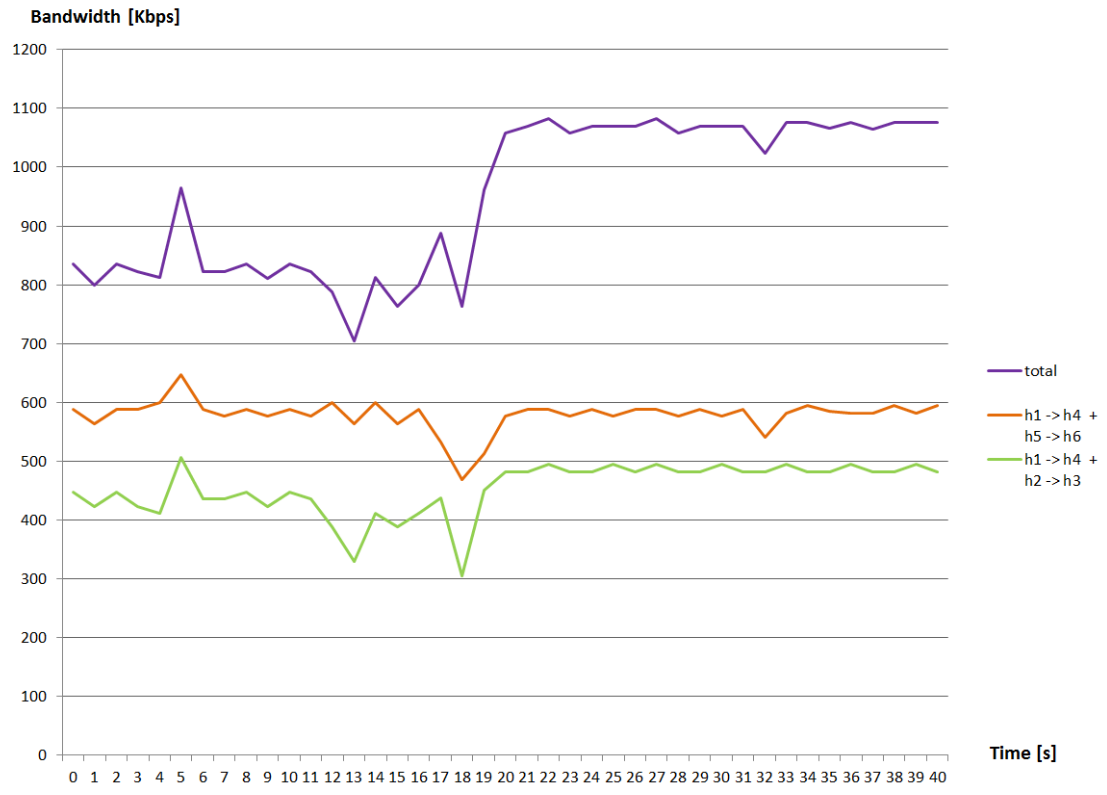
the switches s1 and s2 with the yellow flow from h1 to h4, with both flows providing around 500 Kbps of traffic. Therefore both flows will get around 250 Kbps traffic through this link. Since the blue flow is not involved in any further congestion, around 250 Kbps of traffic should arrive at h3. On the other hand, the yellow flow has to share the link between the switches s2 and s4 with the red flow from h5 to h6. The yellow flow provides 250 Kbps, whereas the red flow provides 600 Kbps. This means for the yellow flow arriving on h4, a bandwidth of around $\frac{250*600}{250+600} \approx 176$ Kbps should be measured and for the red flow arriving on h6 a bandwidth of around $\frac{600*600}{250+600} \approx 424$ Kbps should be measured.
The protocol should detect congestion on the link between s1 and s2 and on the link between s2 and s4. As a reaction of it, the yellow and blue flow are marked from s1 on, and the red and yellow flow from s2 on. Since both, s1 and s2, mark the yellow flow, s2 sends back a feedback packet to s1 informing it about the upcoming congestion. In response to that, s1 starts to lower the priority of the yellow flow. Since this setup uses UDP traffic, the blue flow with the higher priority should completely overtake all the bandwidth of the link between s1 and s2, since it is providing enough traffic to use all the bandwidth on its own. For the measured bandwidth on h4, h6 and h3 we should get 0 Kbps for the yellow flow on h4, 600 Kbps for the red flow on h6 and 500 Kbps for the blue flow on h3 as soon as the prioritization is applied.

The measured throughput for each flow of this test can be seen in Figures 4.2a and 4.2b. Figure 4.2a clearly shows the expected average throughput of 250 Kbps for the blue flow, slightly below 200 Kbps for the yellow flow and around 400 Kbps for the red flow before prioritization is applied. At around 17 seconds, the protocol starts to lower the priority of the yellow flow, resulting in it starving and the blue and red flow overtaking the now available bandwidth. Figure 4.2b shows the sum of all the flows going through a bottleneck, in green the yellow and blue flow with their bottleneck link between s1 and s2, and in orange the red and yellow flow with their bottleneck link between s2 and s4. In both stages, normal scheduling and modified scheduling by the protocol, the link capacity is always at its maximum, around 500 Kbps for green and 600 Kbps for orange. Looking at the total throughput measured at all three receiving hosts combined, it is obvious that some traffic sent along the bottleneck links is dropped in the case of normal scheduling, only achieving an average throughput of 800 Kbps in this case. As soon as the yellow flow sending traffic along the first bottleneck into the second bottleneck gets a lower priority and vanishes, all of the bottleneck bandwidth is used for traffic which reaches its destination, achieving an average throughput of 1100 Kbps. This is an overall increase in performance of 37.5%.

It has to be noted that this test did not use a congestion detecting algorithm and the detection was manually triggered by setting a register on the switch at runtime. One reason for this is the fact that the measurement was done on a different version of the behavioral model of P4, since the version the protocol was developed in did not support changing the packet priority, which was recognized too late. The version used for the measurement has a compatibility issue with the feedback mechanism used when detecting congestion in the egress and informing the ingress by cloning and recirculating a packet, which leads to the switch stopping to work. Another reason is the fact that the implemented congestion detection is extremely unreliable, detecting a lot of false positives. As example, Figure 4.3a shows a simple line network setup. When running traffic through this network from h1 to h3, there should not be any congestion detected on the egress of switch s1, s2 or s3. Despite this, looking at the buffer occupation on the egress of s3 in Figure 4.3b, one can see that the buffer is at its maximum capacity of 50 packets after a few seconds of runtime and remains there for several seconds, which would trigger the congestion detection algorithm. This issue is most likely because the test setup is run in a simulation and would probably not occur when running it in a real network. However, over the course of this thesis this was not tested and the real reason is unknown. Lastly, when doing the measurements, a delay in between the protocol changing the priority of packets and the actual change in throughput measured at the receiving hosts can be noticed. This issue was not investigated further, but it is highly suspected that there is some buffering going on inside the network leading to this delay.

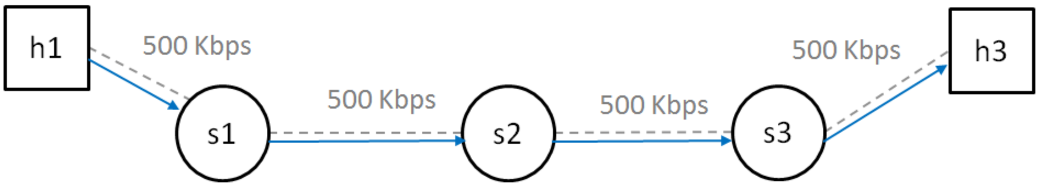


(a) Measurement for each individual flow

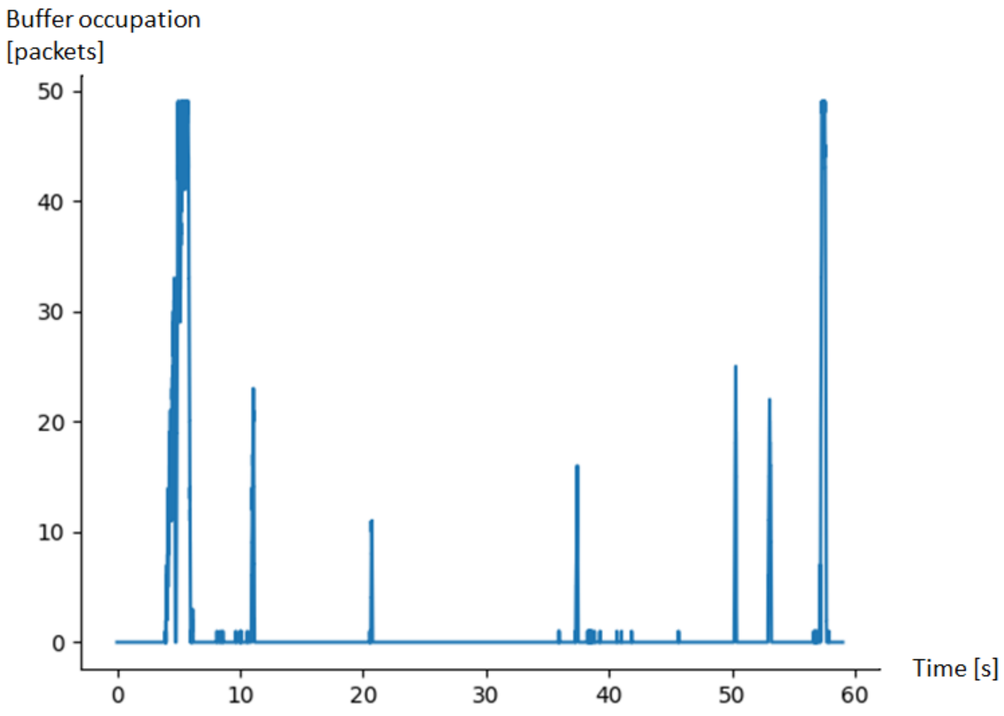


(b) Measurement for the sum of flows

Figure 4.2: Measurements running the test setup



(a) Simple line network



(b) Buffer occupation of switch s3

Figure 4.3: Example showing buffer occupation in a line network

Chapter 5

Future work

While this thesis shows that adapting the scheduling on a switch can lead to a decent increase of the overall network performance, it leaves a few things unanswered that should be addressed in the future. For one, a reliable method to detect congestion is still missing. The implemented version is able to detect congestion, but as shown in Chapter 4 there is also a relatively high chance of detecting false positives. Exploring and testing different approaches as suggested in 3.1.1 needs to be done in order to reduce the chance of false positives and therefore increase the chance of the protocol working correctly and increasing the performance. Furthermore the current version of the implementation does not support any sort of timer, meaning once congestion has been detected the state remains until the network is torn down. A timer is needed to reset the state as suggested in 3.1.4. Additionally switches receiving a marked flow, for which they detected congestion as well, proceed to send back a feedback packet for every single packet they receive. An additional timer, or a meter, should limit the amount of feedback packets that are sent back to prevent unnecessary amounts of traffic as well as a near instant reduction of priority to the lowest value due to the high amount of feedback packets. Last but no least the protocol currently lowers the priority of every single packet of a flow. If there is enough traffic of other flows on a certain egress port, the flow with lowered priority will simply starve. Although this is desired in order to increase the overall performance of the network, other possibilities can be explored to keep routing a minimum bandwidth of flows with lowered priority, still leaving every flow an opportunity to transmit traffic. This would ensure that small flows can still eventually get through even if the network is congested.

Chapter 6

Summary

At the beginning of the thesis the problem was stated that in a congested network, the overall performance can be lowered due to to-be-dropped traffic taking up bandwidth on bottleneck links. By investigating this problem one can find that traffic which is part of a congestion twice along its path, has a high chance to exhibit such behavior. In reaction to that a protocol was proposed to detect these flows and adapt the scheduling on a switch to reduce the amount of bandwidth that gets wasted. Via implementation of the proposed protocol in P4 it was shown that the protocol has the intended functionality. A test case has been set up and measurements of the overall performance of the network showed that using the protocol can increase the performance by up to 37.5% for this case, achieving the maximum possible throughput given by the bottleneck links. This shows that the stated problem can be solved by adapting the scheduling of packets on individual switches based on the information received from the network.

Appendix A

Installation instructions

To rerun the measurements done in Chapter 4 please use the branch vm_test of the code provided in the GitLab. The network can be set up by running

```
$ ./code/switch/p4run_script.sh
```

In a new terminal execute

```
$ ./code/congestion_detection_server.sh
```

which provides the Iperf server setup for the three hosts. Finally in a third terminal

```
$ ./code/congestion_detection.sh
```

provides the three commands to start sending traffic, and a second window in the tmux session provides the commands to trigger the detection on s1 and s2 respectively. In the terminal with the mininet setup there should be outputs when congestion was triggered as well as when a feedback packet arrives (only if digest is enabled).

The master branch provides a version with congestion detection in the egress, but as stated in Chapter 4 this was not used to make the measurements.

Bibliography

[1] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Pfabric: Minimal near-optimal datacenter transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, Aug. 2013.

[2] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 501–521, Santa Clara, CA, Mar. 2016. USENIX Association.

[3] K. Nichols and V. Jacobson. Controlling queue delay. *Queue*, 10(5):20, 2012.

[4] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, Renton, WA, Apr. 2018. USENIX Association.

[5] B. Turkovic, F. Kuipers, N. van Adrichem, and K. Langendoen. Fast network congestion detection and avoidance using p4. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*, NEAT '18, page 45–51, New York, NY, USA, 2018. Association for Computing Machinery.