Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

Control Theory meets programmable Data-Plane

Semester Thesis

Author: Steven Marty

Tutor: Maria Apostolaki, Alexander Dietmüller

Supervisor: Prof. Dr. Laurent Vanbever

September 2020 to January 2021

Abstract

When a network operator changes forwarding rules, depending on some measured network variables, it is often handled in an ad-hoc manner. We introduce a novel system which changes the forwarding distribution with the intend of stabilizing a network variable, as traffic rate, link delay or loss rate, around a target value. This is accomplished with a PID controller, which is beneficial, as it can be tuned depending on the scenario and the operator's need. In this thesis we present difficulties in finding such a reusable controller due to the dynamics of different network variables. Nevertheless, we came up with general guidelines on how to integrate and tune the controller to work as desired.

Contents

Chapter 1

Introduction

1.1 Motivation

There has been a lot of effort made in research to create abstractions to facilitate efficient network telemetry. Network control on the other hand has received less attention and most solution are created in a human-driven manner without providing explicit stability and generality guarantees, which is to some extent justified by the rapidly changing network conditions and the general complexity of a network. Nevertheless, control theory has dealt with complex systems, so this thesis tries to propose a reusable control system, which stabilizes a control variable around a target value on network hardware, meaning by the use of network measurements and network actions.

There exist different scenarios, where a control system can be useful. If a network operator controls a whole network, he can influence many routers and the system is quite complex. The simpler case with just a single router can be modelled much easier and this is the case, our thesis is focusing on: controlling a network variable on a switch only based on the measurement done on the switch itself.

Most of the time, the target value of a switch is to balance some network measure, as traffic rate, along the possible links. There exists already simple approaches like ECMP or packet spraying. For identical paths, equal traffic patterns, uniform flows, meaning the flows get equally distributed by the hash function, and no external disturbances, this would be enough and work as intended. But due to those problems mentioned, equal traffic distribution may not be achieved in the case of big flow size disparities or it may not even be desirable to equally distribute, as one path might be congested. Therefore, we propose a controller that handles the forwarding behaviour of flows and changes it depending on some measured network variables.

1.2 Task and Goals

To verify the proposed controllers, we simulate a simple topology with legitimate looking traffic. With this setup, different controllers for different network measures as process variables can be evaluated. The first network measure that comes to mind in this context, is the traffic rate. The goal is to have both links filled with an equal amount of traffic, even if there is an inequality of flow sizes or certain flows reduce their rate due to TCP.

Another network property that an operator might like to achieve, is the avoidance of congestion or having equal delay and packet loss along the possible links. We present a controller, which tries to balance delays and forwards traffic along the faster link, as long it is not congested and its delay would increase. Furthermore, controlling packet loss rate and stabilizing it between the links is the

last case proposed by this thesis. Both those controllers, for delay and packet loss, pursue the same goal of avoiding or minimizing congestion.

The thesis tries to identify the networks processes and their dynamics. Based on that, we suggest general tuning rules for the controllers, which should make them easy to integrate in a new network with new properties, without tuning them much by hand, but having a guideline about what to take care of. The thesis also reveals difficulties in using control theory for network processes.

1.3 Overview

In Chapter 2, we introduce relevant theory about control and mention a few papers, which are also measuring a network variable and change their forwarding behaviour depending on the measurement. The next part, Chapter 3, presents the controller and measurement algorithm proposed in this thesis. Chapter 4 evaluates the controller in different scenarios and under different conditions. Chapter 5 briefly discusses the findings and wraps up the thesis.

Chapter 2

Background and Related Work

In this chapter, we describe some theory, to establish the ground for the thesis, and present important related work.

2.1 Background

2.1.1 Control Theory

Control Theory deals with controlling dynamic systems like machines or other processes. In this thesis control theory and its approaches are applied to computer networks.

A control system contains a reference signal $r(t)$, an input signal $u(t)$, the output signal $y(t)$ and the system dynamics which links input to output (Figure 2.1). The controller calculates the input signal for the system. One differentiates between open-loop and closed-loop controller. The former does not take the output $y(t)$ into account for calculating the controller input. The latter, used in this thesis, subtracts the output $y(t)$ from the reference $r(t)$ to get the error signal $e(t)$. This error signal is fed through the controller to determine the optimal input, which drives the system to the reference point.

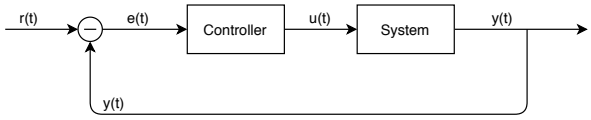

Figure 2.1: Closed Loop Controller

2.1.2 PID Controller

One of the most widely used controller is the PID controller, which calculates the input signal based on the error signal itself (P), the integration (I) and the derivative(D) of it (Equation 2.1) [2]. K_p, K_i and K_d represent the respective gain of each component. Instead of using K_i and K_d, there is a standardized form with τ_i and T_d (Equation 2.2), which will be used in this thesis.

$$u(t) = K_p e(t) + K_i \int_0^t e(t')\ dt' + K_d \frac{de(t)}{dt} \tag{2.1}$$

$$u(t) = K_p\Big(e(t) + \frac{1}{\tau_I}\int_0^t e(t')\ dt' + T_d\frac{de(t)}{dt}\Big) \tag{2.2}$$

Numerous papers have been published about tuning the PID parameters, but the main roles of them are the following. When a step input is applied, the system has settled (steady-state) and there is an offset between the desired and actual value, the integral part is able to correct it. The objective of the derivative part is to damp the input and reduce overshoot. Overshoot can result in oscillation because if the controller tries to correct the overshoot by applying an input too large in the other direction, it will overshoot in this direction and starts to oscillate.

Calculating the input with the PID controller and applying it to the system seems quite straightforward, but one can not simply apply any input because of system limitations. The controller may calculate an input which is not possible due to physical properties or restrictions. So the system will just apply maximum input and if there is still an error, the integrator will add up this error, it "winds up", and the controller just goes on setting the maximum input, independent of the output. When the system comes back to a normal state, the controller has accumulated a lot of error for which it is then required to have an error with the opposite sign for a long time to cancel it out. This must be prevented.

2.1.3 System Parameters

For tuning the controller, it is beneficial to have a parametric model of the system, which describes the dynamics, meaning the relation from input to output. Unfortunately, obtaining these dynamics analytically is often quite difficult, but some information about the system can be extracted by examining the step response [2]. The step response is measured when a step input is applied to the system. From the response, one can obtain the gain, rise time, overshoot, settling time etc. The linear time-invariant system that is used in this thesis to model the process, is a first order plus dead time system:

$$\tau_1\frac{dy(t)}{dt}y = -y(t) + ku(t-\theta) \tag{2.3}$$

The dynamics of the system can be described with these three parameters:

- **Plant Gain** k is the ratio of the input and the output when steady state is reached $\frac{\Delta y}{\Delta u}(t \to \infty)$.
- **Dominant Time Constant** τ_1 is the time until the output has moved 63% of the way to its new steady state.
- **Dead time** θ is the time it takes until the output moves in the right direction.

Figure 2.2 shows how those values can be extracted from the step response. With these parameters, PID tuning rules can be applied [5]. This approximation only works if the system is stable, meaning the output is bounded for a bounded input.

2.2 Related Work

2.2.1 PID Tuning

Sigurd Skogestad proposed, next to many other papers, analytical PID tuning rules that are simple, but still result in a good closed-loop behaviour [2]. The first step of tuning is identifying the system

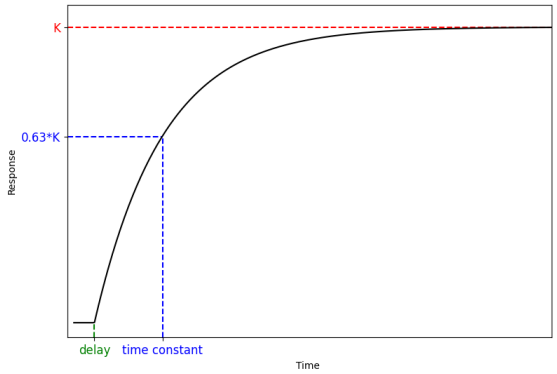

Figure 2.2: First order plus dead time step response

with its parameters as it is described in Section 2.1.3. Once the system is identified, the parameters can be put in the following equations to get the gains:

$$K_c = \frac{1}{k}\frac{\tau_1}{\tau_c + \theta} \tag{2.4}$$

$$\tau_I = min\{\tau_1, 4(\tau_c + \theta)\} \tag{2.5}$$

where $\theta < \tau_c < \infty$ is the tuning parameter. The optimal value for it is a trade-off between fast speed and stability. He proposes to use $\tau_C = \theta$ for a robust system.

2.2.2 Control in Networks

Other papers have been published, which propose a system that measures a network variable and changes the forwarding behaviour, based on the monitored variable.

Measuring the current link utilization with a DRE, a discounting rate estimator, by only using one register, was proposed by Alizadeh, Mohammad, et al in their system named CONGA [1]. A register X is incremented for each packet sent over a link by the packet size in bytes and is decremented periodically (every T_{dre}) with a multiplicative factor α between 0 and 1: $X \leftarrow X \cdot (1 - \alpha)$. This algorithm is similar to the exponential moving average mechanism, but requires only one register and reacts quickly to traffic bursts. They use this signal for congestion aware load balancing.

Holterbach, Thomas, et al goal is to detect failures and reroute afterwards. They came up with a system called Blink [4], which is based on TCP re-transmission signals. The system monitors TCP sequence numbers and looks for consecutive re-transmissions, as it tries to detect failures.

Chapter 3

Design

This chapter focuses on the general controller design and how it is tuned, along with some controller modifications.

3.1 Controller

This thesis proposes a controller, which splits flows across the possible links, based on some network measure one wants to balance. Initially, we consider only two forwarding links. In static ECMP the 5-Tuple of a flow gets hashed and the hash value modulo the number of paths corresponds to the forwarding link to equally distribute the traffic. To control the traffic more precisely, a system is needed, where the preferred route distribution can be controlled. Therefore, we introduce a range of values, which we the call hash array. After the 5-Tuple hash value is calculated, we apply the modulo operation with the size of the hash array to get a value within this range. For now, we take the range from 1 to 100 as our hash array. To arrive at a system, which behaves like normal flow-based ECMP, the hash would be calculated and if it is below 50 link 1 would be taken and if it is higher it would be link 2:

$$h = Hash(\text{5-tuple}) \mod 100 \tag{3.1}$$

$$\text{Outgoing Link decision} = \begin{cases} 1 & 1 \leq h \leq 50 \\ 2 & 50 < h \leq 100 \end{cases} \tag{3.2}$$

It is now possible to shift the flow distribution by increasing or decreasing the decision value (50). From now on, this value is referred to as the hash index. By modifying the flow distribution, we also hope to change the rate distribution. Depending on the number of flows, their starting times, disturbances and the non-uniform distribution of the flows, this relation can fluctuate. Fortunately, since we use a closed-loop controller, it gets feedback of the actual network measures and can adjust to it.

Our controller receives an error signal, based on some network measures, and changes the hash index accordingly. The calculated hash index is $u(t)$, the network measure one wants to reach is $r(t)$ and the actual network measure is $y(t)$, resulting in the error signal when subtracted from each other (see Figure 2.1).

In the scenarios simulated in this thesis, the primary goal is to balance certain network measures along the two links. The network measures investigated are traffic rate, link delay and loss rate. To summarize the general approach, the measurement for link 1 is called $m_1(t)$ and $m_2(t)$ for the link

2. As we want to balance these two, we could set the controlled variable as the difference between the two (Equation 3.3) or the ratio between the two (Equation 3.4). To achieve equal measures, the reference $r(t)$ would need to be 0 or 0.5 respectively.

$$y(t) = m_2(t) - m_1(t) \tag{3.3}$$

$$y(t) = m_2(t)/(m_2 + m_1(t)) \tag{3.4}$$

In this thesis, we chose the first, due to the thought, that if one measurement is twice as big as the other, for example, one link has twice the delay of the other, the case where the delays are small, is less severe compared to the case when delays are large. This means an imbalance of paths having 50 ms and 100 ms delay is less concerning than 400 ms and 800 ms delay. Nevertheless, one could also choose the second one and most of the ideas would apply.

In certain network scenarios, the measurements, thus also the error signal, can have arbitrary values. Picking the parameters of the controller would therefore depend on the setting. But our proposed controller should be easy to use in different scenarios by defining just some network properties. Hence an error signal, which always lies in the same range, would be beneficial. This can be accomplished by normalizing it, such that $m_1(t)$ and $m_2(t)$ lie between 0 and 1. Consequently, $y(t)$ ranges from -1 to 1.

We only consider the proportional and integral part of the error, so in fact, only a PI controller is used. This thesis omits the differential part because of its sensitivity to noise. The PI controller will output a continuous value, but the hash index is an integer, so it must be rounded. Furthermore, the output of the controller is multiplied by the hash array length l to separate things a bit more. If one would need finer control by using a bigger range or very few flows are sent and a smaller range is enough, the same controller can be used.

If we assume that the default behavior is similar to ECMP, meaning splitting the traffic in half, it makes more sense using the controller's output as the offset from the middle of the hash array, opposed to setting it as the hash index directly. The latter would still work, as after time has passed with a high error signal, the integrated error will push the hash index to the optimal value. In consequence, the hash index is set according to Equation 3.6.

$$u(t) = K_p\Big(e(t) + \frac{1}{\tau_I}\int_0^t e(t')\ dt'\Big) \tag{3.5}$$

$$\text{hashindex} = round\Big(u(t) \cdot l + \frac{l}{2}\Big) \tag{3.6}$$

To counteract integral windup, integral clamping is used. This ensures, that if the error signal is so large that it would exceed the hash index range, it is not added to the integral.

The entire theory from Section 2.1 and the ideas presented in this chapter are specified for continuous systems. In digital devices, such as switches or routers, you normally encounter discrete systems as a value is measured every interval T. Theory about continuous control can still be applied, as these systems are not inherently discrete, but merely because we sample from them every T.

When sampling traffic rate, loss or delay, one experiences quite noisy measurement for certain conditions. Noise can be reduced by choosing a higher T. Alternatively, the variable can be measured with a moving average window, to smooth out short-term fluctuations. The duration of this moving average time window will be called W. This way, we can keep the controller's update frequency high, while still having stable rate measurements. T is therefore just the controller input

interval and W the measurement window. T can be selected based on hardware limitation and W should be large enough to have a more or less stable signal. The downside of a large W is, that incidents may seem less intense as the measurement is averaged over times when everything was normal, so it slows down the response. As a consequence, the system may accumulate error, because the output variable does not change fast enough after an input change. This results in overshoot and oscillation, so W can not just be chosen as large as needed for a stable signal, because it involves other consequences.

It is also possible to use an exponential moving average which adds more weight to the most recent measurement, while still smoothing out the signal. This would not slow down the system as much. If one has memory restriction, some measurements could be implemented with the DRE from CONGA [1], which, due to its fast reaction to bursts, has also a fast system response (explained in Section 2.2.2).

3.2 Tuning Procedure

The topology is set up, the measurements are available and we have a controller at hand that tries to drive the system to the desired state. One question remains open, how should the controller parameters K_p and T_i from Equation 3.5 be set? This depends on what network variable we want to control, but in general, one gets a good idea of the dynamics when analysing the step response. This is done by applying a step input, in our case an increase of the hash index and observing the dead time and the time it takes to reach the steady-state. If the system is linear, one can identify the system as in Section 2.1.3. If the system is non-linear the tuning becomes more of a hand-made approach, whereas for linear systems general tuning rules as in Section 2.2.1 exist. Those rules can not be used straightforwardly, as the measured variable can have much noise which will be amplified by high controller gain, causing undesirable behaviour. In the next chapter, we will address these problems.

3.3 Variation

3.3.1 Multiple Links

How can these ideas be used for more than two links? For three links we can still use one hash index range but with two hash indices (h_1 and h_2). The forwarding rule would be as the following (l is the length of the hash array):

$$h = Hash(\text{5-tuple}) \mod l$$

$$\text{Outgoing Link decision} = \begin{cases} 1 & 1 \leq h \leq h_1 \\ 2 & h_1 < h \leq h_2 \\ 3 & h_2 \leq h < l \end{cases} \tag{3.7}$$

For both the hash indices, a PI controller is used, which tries to balance the network variable between the two links it has control of. This measns that h_1 is responsible for link 1 and 2 and h_2 for link 2 and 3. The hash index calculation from equation 3.6 must also be adapted. Instead of scaling the error signal to the whole hash range, each controller scales it to its available range, resulting in:

$$h_1 = round\Big(u_1(l * h_2) + \frac{h_2}{2}\Big) \tag{3.8}$$

$$h_2 = round\Big(u_1(t)(l - h_1) + \frac{l + h_1}{2}\Big) \tag{3.9}$$

3.3.2 Multiple Network Measures

The controller can easily be applied to control multiple network measures together. This is possible, since all the network measures are scaled to the range from 0 to 1. Each network variable has a different weight w_i, resulting in a final signal, in which m_j^i is the signal for variable i and link j:

$$y(t) = \sum_{i=1}^{n} w_i(m_2^i - m_1^i) \tag{3.10}$$

$$\sum_{i=1}^{n} w_i \stackrel{!}{=} 1 \tag{3.11}$$

This way $y(t)$ is in the same range as in the simple case with one variable and therefore, we can use similar controller parameters. Nevertheless, one has to be careful if some values have completely different dynamics. An example of this idea is evaluated in Section 4.5.2.

Chapter 4

Evaluation

In this chapter, we cover how to use a controller for different network scenarios with different process variables. In the different scenarios we try to achieve equal link utilization or avoiding congestion by balancing delay or loss between the two output links. But first, the implementation and the general network settings are explained in Section 4.1.

4.1 Experimental Setting

4.1.1 Implementation

The network topology, the traffic and the controller are implemented in *ns3*[1], a widely used discrete-event network simulator built in C++ and Python. In general, there is a main script which sets up the topology with the corresponding delays, bandwidths, queues, etc. In the main script, we also assign controller and measurement procedure to the nodes as well as defining the traffic pattern and the disturbances. We use the *ns3* application *BulksendApplication*[2] for sending TCP flows. The distribution of start times and flow sizes will be explained in the next Section. The basic controller and measurements are implemented in a module, which can easily be included in future scenarios[3].

A traffic queue, with the size of the bandwidth-delay product, is installed on each outgoing link. The delay chosen for calculating the queue size is the maximum RTT a node experiences. The queue system configured in the simulation is a simple First in - First out queue. When setting it up in *ns3*, one has to remember that on top of these link queues, each node has an additional traffic queue installed by default[4]. We removed this queue, to better simulate a router or switch behaviour, as normally there is also only one queue.

4.1.2 General Network Scenario

The general setting consists of a network with four nodes, one sender and one receiver (Figure 4.1). The first node has two equal-cost paths to the receiver.

We simulate network traffic using a Poisson process to determine flow inter-arrival times [3] and empirical distributions to determine flow sizes. So the inter-arrival times $A_n = T_n - T_{n-1}$, where

[1] `https://www.nsnam.org/`

[2] `https://www.nsnam.org/doxygen/classns3_1_1_bulk_send_application.html`

[3] `https://gitlab.ethz.ch/nsg/students/projects/2020/sa-2020-55_control_theory_meets_data_plane/-/tree/master/workspace`

[4] `https://www.nsnam.org/docs/tutorial/html/building-topologies.html#queues-in-ns-3`

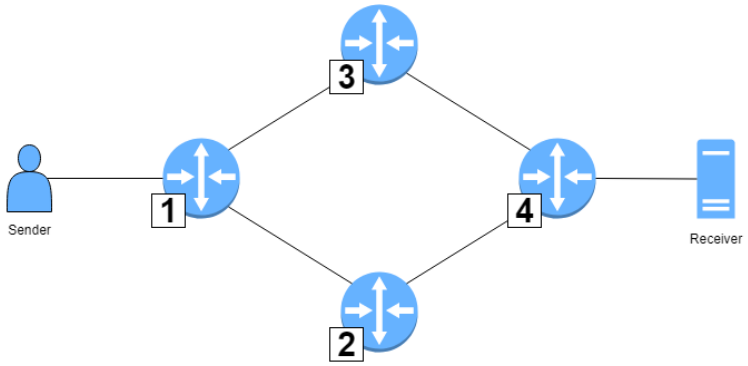

Figure 4.1: Topology

T_n is the arrival time of flow n, are exponentially distributed with a rate parameter λ:

$$P(A_n \leq t) = 1 - e^{-\lambda t} \tag{4.1}$$

The mean inter-arrival time is λ. Figure 4.2 displays an example of starting times for new flows, where the time between the start of two consecutive flows are samples of distribution 4.1 with $\lambda = 2ms$.

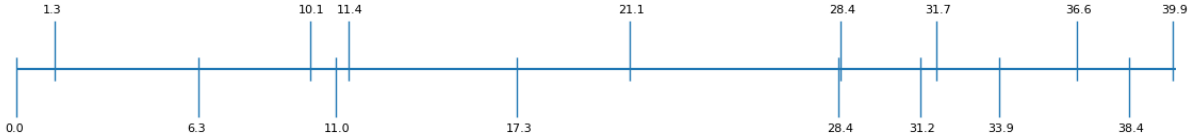

Figure 4.2: Samples of the arrival times T_n of flows with a Poisson process

Flow size distributions are more complex to model than the inter-arrival times. Fortunately, there exist empirical CDFs of flow sizes [5] we can use. In the experiments we sampled from the *Fabricated_Heavy_Middle* distribution (see Figure 4.3). With this flow size distribution, λ can be calculated based on the desired traffic rate:

$$\lambda = \frac{\mathbb{E}[\text{flow size}]}{\text{traffic rate}}$$

4.2 Controlling Traffic Rate

4.2.1 Scenario

The goal is to have equal amount of traffic over both links. Just using ECMP, can still result in an imbalance, due to temporal flow size variation or non-uniformly distributed flows (see Section 1.1). To see this issue, 100 equal size flows are simulated sending traffic from the sender to the receiver where all links from the network in Figure 4.1 have equal bandwidth of 1 MBps. The flows are not perfectly split in half (see Figure 4.4). This mismatch can be solved by utilizing a controller.

The traffic rates, needed by the controller, are simply measured by counting all the bytes sent over the link and dividing it by W, the measurement time window, every interval T (Equation 4.2).

[5] `https://github.com/PlatformLab/HomaSimulation/tree/omnet_simulations/RpcTransportDesign/OMNeT%2B%2BSimulation/homatransport/sizeDistributions`

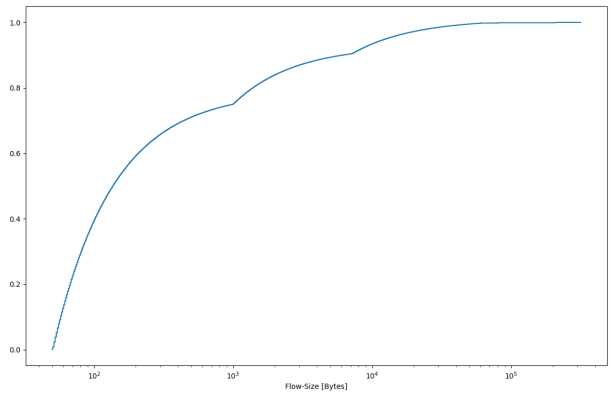

Figure 4.3: The flow size distribution used in this thesis

This measurement is normalized by dividing it with the maximum observable bandwidth to get a signal between 0 and 1. The process variable $y(t)$ is the difference between the two scaled traffic rates.

$$y(t) = \frac{bytes_2 - bytes_1}{W} \frac{1}{max\_bandwidth} \tag{4.2}$$

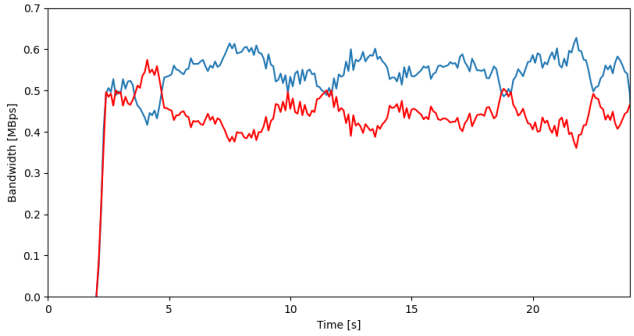

Figure 4.4: 100 flows not being equally distributed along the two links by using ECMP. This is due to the burstiness of packets and not splitting the flows exactly in half.

4.2.2 System Identification

To model the process as a first order plus dead time system, the three parameters from Section 2.1.3 (k, τ_1, θ) are required. Our measurement system has some dead time θ as we measure the rate only every T and until then, no changes in the output can be seen. The system itself also has a dead time, but this is rather small because as soon as the control input is applied, the traffic is distributed according to the new hash index. It takes only one packet of a flow with a hash value in the changed hash index interval Δh to arrive at the node. With a bandwidth of b, a hash array of length l, an average packet size of p and the assumption of equal distribution of flows on the whole hash array the expected time is:

$$\mathbb{E}[\text{Dead Time}] = \frac{p}{b}\frac{l}{\Delta h} \tag{4.3}$$

For a link with 1 MBps bandwidth, an average packet size of 500 bytes, a hash array of length 100 and a hash index change of 20 the dead time would be 2.5 ms. So in general, the measurement interval T, which was generally set as roughly 50 ms, is mostly accountable for the dead time.

The time constant τ_1 is the time it takes until most flows, which lie in the new hash index interval, pass the switch. Not only this adds to the time constant, but also the fact that for measuring a moving average is used. So when the hash index is changed, the controller still includes packets prior to the change in the new rate measurement, thus the system takes some time until we are able to see the change in the output.

In the end, both these parameters (θ and τ_1) are mostly dominated by the rate measurement parameters (W and T). This indicates that broadly speaking, we can define the system dynamics ourselves by choosing T and W. The rate measurement at time t depends on the data in $[t - W, t]$ so it will take W to see the full steady-state response. This means, to see 63.2 % of the steady-state response it takes $0.632 * W$ time and we get $\tau_1 = 0.632 * W$ as well as $\theta = T$ as explained above.

For calculating the gain, we imagine sending traffic with the hash index set to 0 ($u(t) = -0.5$) and we would measure $y(t) = -1$. When applying $u(t) = 0.5$ (resulting in the maximum hash index) we expect $y(t) = 1$. In plain language this just switches all the traffic from one link to the other. So the gain is $k = \frac{\Delta y}{\Delta u}(t \to \infty) = \frac{2}{1} = 2$.

We confirmed these analytic values in a simulation (Figure 4.5), where a step input is applied at 5 seconds ($u(t \leq 5) = -0.5$ and $u(t > 5) = 0.5$). The τ_1 of the system is the time it takes to reach 63.2 % and as described above, it is $0.632 * W$ rounded to the next T for all the cases and the dead time corresponds to T.

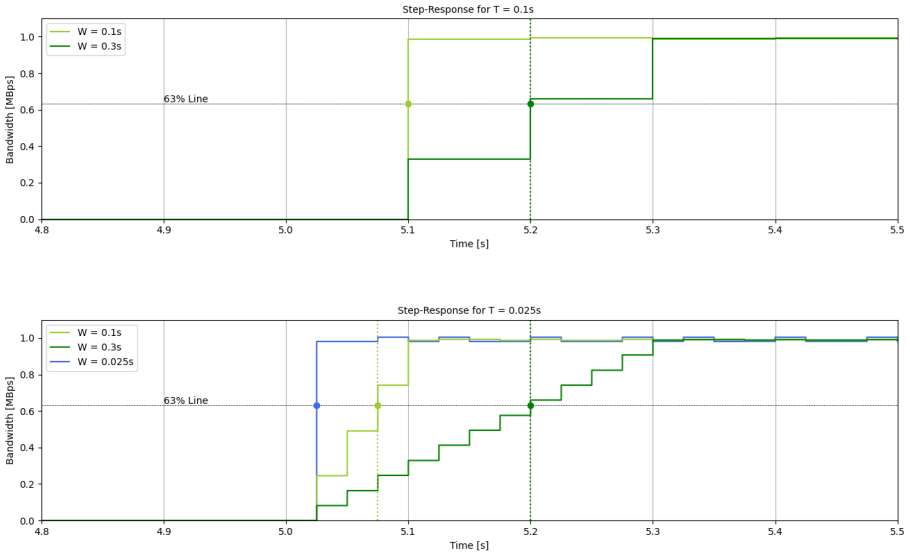

Figure 4.5: Step response for control input $u(t \leq 5) = -0.5$ and $u(t > 5) = 0.5$ with different T and W, confirming the analytical values.

4.2.3 Tuning

Theoretically, the system is linear time-invariant as the process variable (traffic rate) is proportional to the applied input (hash index) when the steady-state is reached. Additionally, the process variable does also not depend on time. This can be verified by looking at an example where the hash index is increased from 20 to 60 (Figure 4.6), confirming that the output is proportional to the input (with some noise).

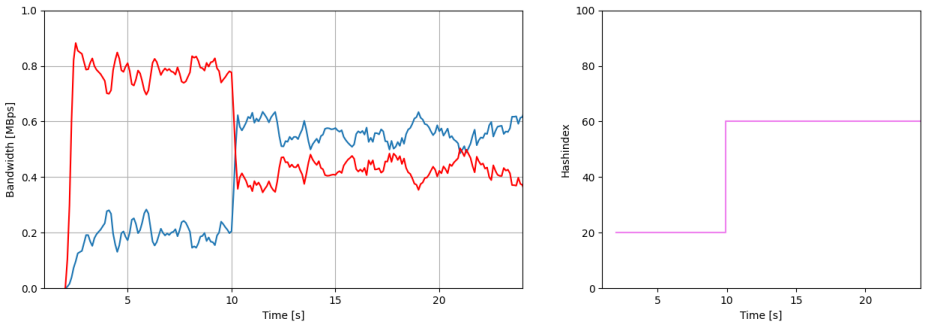

Figure 4.6: The hash index is changed from 20 to 60 after 10 seconds, showing that the system is indeed linear, because the rates are proportional to the hash index.

Since the system is linear, we can apply the tuning rules from Section 2.2.1. Plugging in the values will give us these controller parameters, based on τ_c, which is chosen depending on stability and speed requirement:

$$K_p = \frac{1}{k}\frac{\tau_1}{\tau_c + \theta} = \frac{1}{2}\frac{0.63 * W}{\tau_c + T} \tag{4.4}$$

$$\tau_I = min\{\tau_1, 4(\tau_c + \theta)\} = min\{0.63 * W, 4(\tau_c + T)\} \tag{4.5}$$

Setting $\tau_c = \theta$, in our case $\theta = T$, should give a reasonably fast response with moderate input usage and good robustness margins (see section 2.2.1). Increasing τ_c leads to slower, but also less oscillating controller responses (Figure 4.7). In these plots, the reference is changed and the controller reacts to it. This gives a good picture of the controller's dynamics. The controllers in these examples have a T of 0.1s and a W of 0.3s. In this scenario, the proposed $\tau_c = \theta$ is too aggressive as the measurement is noisy. An aggressive controller will force many unnecessary hash index changes and consequently, routing packets of some flows over different paths which can result in reordering at the receiver. By setting τ_c higher, one gets a less aggressive controller. Unfortunately, based on numerous measurement, there is not just one τ_c that works for every window size W and measurement interval T.

In general, we want to have a proportional gain, depending on the window size, because a bigger window gives a more stable measurement, so the value of the signal is more meaningful. Second, we want some reasonable integral gain. The integral gain is $\frac{K_p}{\tau_I}$ and from Figure 4.7 0.8 seems adequate (middle plot). So we choose τ_c to decouple the proportional gain from the controller interval T as we only want W to influence it. On top of that, the integral gain should be 0.8 independent of the window. This results in the following rules:

$$\tau_C = 0.6 - T \tag{4.6}$$

$$K_p = \frac{1}{2}\frac{0.63 * W}{0.6} \approx \frac{1}{2}W \tag{4.7}$$

$$\tau_I = min\{0.63 * W, 0.24\} \tag{4.8}$$

$$\frac{K_p}{\tau_I} = \frac{W}{2 \cdot 0.63W} \approx 0.8 \tag{4.9}$$

The integral gain is roughly 0.8 (see Equation 4.9), as long as the window is smaller than 0.24. If it is bigger, the integral gain will increase. Choosing a W too large is not beneficial as the system takes very long to react, so one should stay in the lower region.

So how should the measurement parameter W be selected? It should be chosen based on the traffic volume, to obtain a measurement without a lot of noise. Suppose, in expectation we want to measure a packet for every hash value in the whole range for one measurement. To achieve this, W has to be at least

$$\frac{p \cdot l}{b}$$

where b is the bandwidth. As packets for a certain flow often arrive in bursts, it is preferable to take five to ten times this value. In fact, it is best to just measure the rate and check how large W should be to get a good measurement, as traffic patterns are very variable.

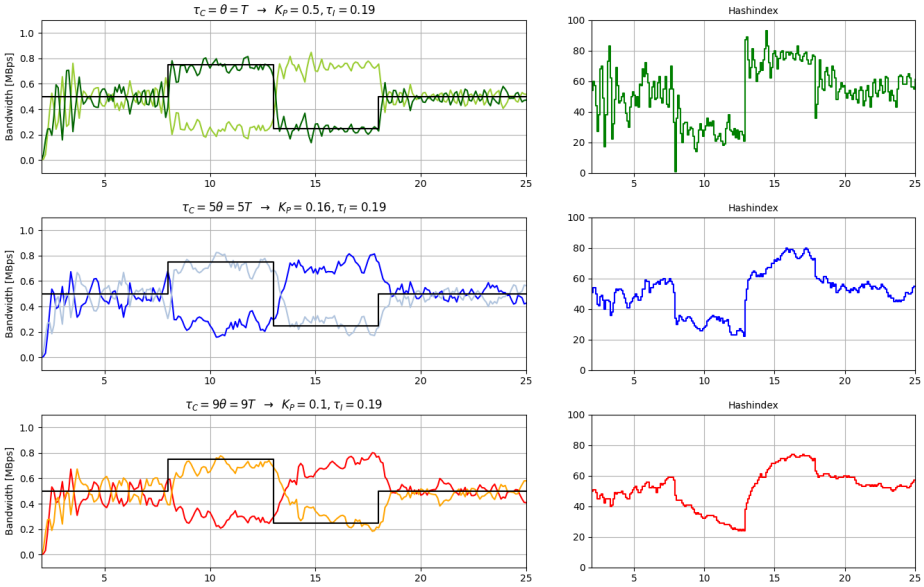

Figure 4.7: The setpoint is changed, black line, and the controller adapts to it. This shows the dynamics of the different controllers for $\tau_c = \theta, 5\theta, 8\theta$. Increasing τ_c leads to slower, but also less oscillating controller responses.

One may notice, that all guidelines presented, work for the scenario with 1MBps traffic. So for a general tuning rule, one has to introduce some scaling based on the expected traffic rate. If there is less traffic, we would choose a bigger W proportional to the traffic and consequently also increasing K_p (see Equation 4.7), which is not necessarily desirable as $y(t)$ is scaled to be always between -1 and 1. An approach would be to have a τ_c depending on the traffic rate, but since τ_c can only influence τ_I when τ_1 is bigger than $4(\tau_c + \theta)$, this is only partly possible. So, we found determining the optimal τ_c to be more challenging than tuning K_p and τ_I directly. We present our insights below:

- K_p**:** If W has been set properly, a value of K_p between 0.05 and 0.3 is reasonable. The noisier the traffic, the smaller the K_p.
- τ_I**:** The integral gain is $\frac{K_p}{\tau_I}$ and this value should be between 0.6 and 1 depending on how fast the controller should react. Higher value results in faster reaction.

4.2.4 Controller in Action

The controller is tested during certain scenarios and disturbances. The simplest case is to just counteract the non-equal flow distribution along the links and temporal differences. In another scenario, one of the two links has loss, which slows down TCP connections.

Non-uniform flows

To see the effect more clearly, less legitimate traffic is sent, as randomness in flow sizes and inter-arrival times can help the fact that flows are not split exactly in half. By just starting 100 big flows at the beginning this is better visible (Figure 4.8). With the simple ECMP algorithm, one link experience more traffic and due to bursts, temporal differences between the rates can be seen. The controller counteracts the non-optimal flow distribution with an average index less than 50. On top of that, it also equalizes the temporal differences.

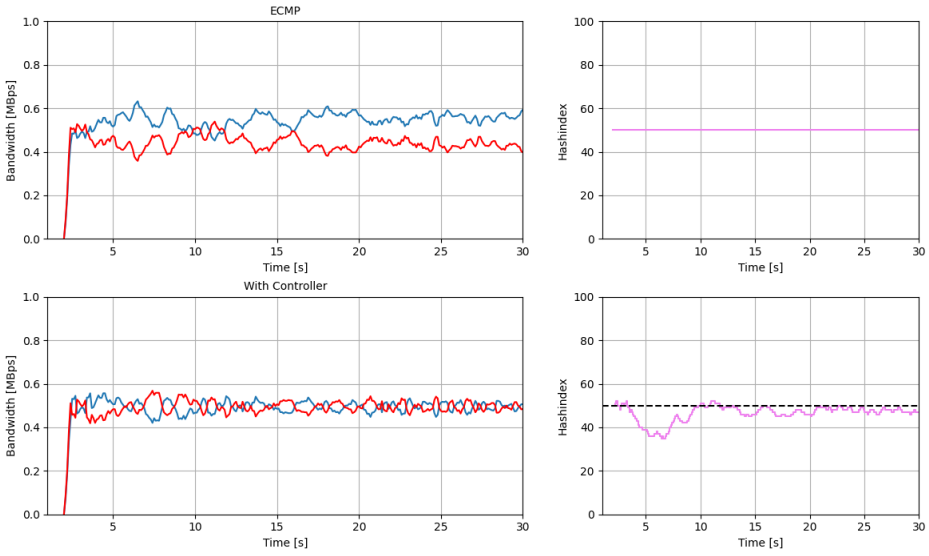

Figure 4.8: 100 flows start sending traffic and due to the non-uniform input, the hash function does not distribute flows equally. Additionally there are temporal differences due to bursts, so the rates sent over both links are not equal. The controller equalizes both of these issues.

Loss

The TCP congestion control algorithm interprets loss as a signal for congestion and reduces the sending rate when loss is encountered. The cause of packet loss may be far away from our network. Nevertheless, the link with flows experiencing loss, does not have the same traffic rate as the other one and the node could actually send more flows over that link. To see this issue, in addition to

the "legitimate" traffic, 25 big flows start sending traffic from the beginning. The small flows will not really react to loss as they are terminated very quickly. At the start, there is only an imbalance due to temporal differences, but at t=10s loss is introduced in one link, slowing down some of the big flows. The controller restores the balance. After roughly 20s, the big flows are finished sending traffic and the loss does not affect the rate sent by the sender as drastically as before, so the hash index goes back to 50 (Figure 4.9).

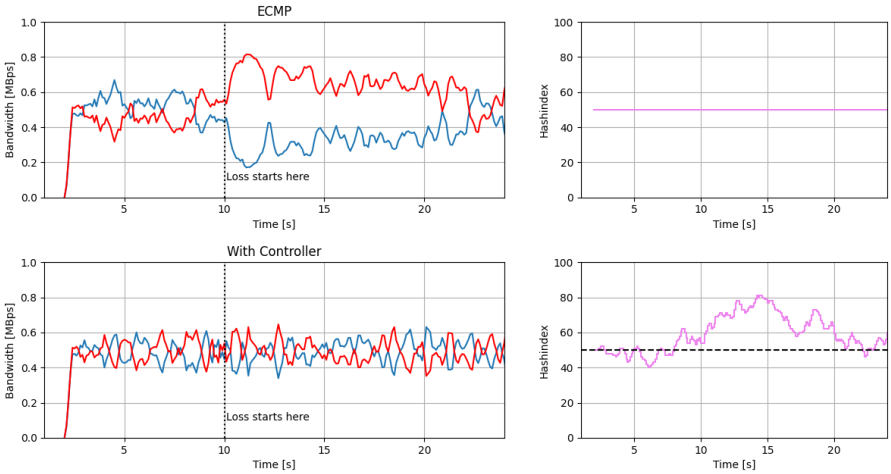

Figure 4.9: Legitimate traffic with some big flows adapt to the loss (on one link) which starts at 10s. Therefore they reduce their rate, creating an imbalance between the two rates of the links. The controller adapts the hash index accordingly and after the big flows are finished, the loss does not influence the small flows as much and the hash index returns to 50.

4.3 Controlling Link Delay

4.3.1 Scenario

In this scenario, we control the transmission delay of two links. To approximate the link delay, only the delay of the handshake, namely the time until a SYN ACK packet is returned after the SYN is sent, is monitored. Focusing only on these two packets makes the task of measuring link delay easier. The signal could be improved by monitoring the other packets as well, but due to reordering and packet loss, it would make things more complicated. One has to be careful when the SYN is lost and a new handshake is initiated after a timeout, as we do not want to measure the timeout. So lost packets are not considered in this signal.

To have a reasonable signal, many flows must start during a measurement interval W. Otherwise, one could regularly send ping messages to get delay signals of the link. In the simulation presented in this thesis, enough flows are starting during a time interval, so this is not needed. The topology is the same as in Figure 4.1 from Section 4.1.2. The goal is to distribute the traffic along the links to minimize the delay difference between them. This can mean that sending everything over the faster link if this is possible or taking away traffic from a link that is congested and therefore having full queues and a bigger delay. If both links have an equal state, ECMP would do fine, but during congestion, there is only TCP that can help, by just reducing the rate of the flows, which is not necessarily desirable as an operator if there is another empty link available.

In both links, the delays for the new flows are measured and averaged every W, summarizing the state of the link j.

$$y_j(t) = \frac{\sum_{i=1}^{n_j} d_j^i}{n_j} \tag{4.10}$$

This signal can have any arbitrary value depending on the network and its state. Hence, it is beneficial to normalize to a value between 0 and 1, 1 meaning full congestion and 0 meaning empty queues. There are two reasonable choices for normalization. The delays are either normalized by dividing by the maximum possible delay of both the links or by doing it for both links separately. The first case really tries to minimize the absolute delays difference, whereas the latter is a way of just minimizing congestion between the two links, as the signal is scaled proportionally to the link properties. This thesis focuses on the first case, but most of the ideas can be applied to the second one. By just scaling with the maximum we get a signal between 0 and 1 indeed, but some values between 0 and 1 are not possible as there is a minimum delay per link (propagation delay). To utilize the full range from 0 to 1 and just measure queue delay, the minimum delay is first subtracted before dividing it by the difference of maximum delay and minimum delay (range of possible values).

$$y(t) = \frac{y_2(t) - \text{min delay}}{\text{max delay} - \text{min delay}} - \frac{y_1(t) - \text{min delay}}{\text{max delay} - \text{min delay}} \tag{4.11}$$

$$y(t) = \frac{y_2(t) - y_1(t)}{\text{max delay} - \text{min delay}} \tag{4.12}$$

It can happen that during a time interval W no handshake has been captured. This could be due to randomness (no flow starting) or due to fully congested links. For this scenario, we assume the first case and just use minimum delay for this interval.

In the setting where both links have equal bandwidth, big enough to hold all the incoming traffic, each link experiences the same delays independent of the hash index. One can create a difference by filling one link with additional traffic or just reducing its bandwidth. To simulate it, we change the scenario such that one link has a bigger bandwidth than the other (the links from node 2 to 4 and 3 to 4). This way the task of the controller is to distribute the traffic, such that the links are not congested.

4.3.2 System Identification

The dynamics are quite different compared to the traffic rate system (Section 4.2). The easiest way to see the difference is analysing an example. 1 MBps traffic arrives at node 1, which needs to be distributed along a link of 0.6 MBps and a link of 0.4 MBps. Setting the hash index to a certain value, for example 20, has resulted in measuring about 20 % of the traffic over the first link. In this case though, one link will just fully congest, reaching the maximum delay, whereas the other one has minimum delay, as it receives less traffic than its bandwidth. The same would happen if the hash value is set to 30, as still, one link would be fully congested. This demonstrates the non-linearity of the process. Figure 4.10 verifies this in a similar example, where the hash index moves from the "optimal" value to 40 and later to 70, resulting in a state with similar delays and states where one of the links experiences maximum delay.

Consequently, this system is neither stable nor linear and can therefore not be modelled like the one with traffic rates. Nevertheless, the dynamics can still be analyzed.

The time it takes until changes in the input are seen in the output is dominated by the system itself and not the measurement. This is due to the fact, that applying a hash index change of Δh

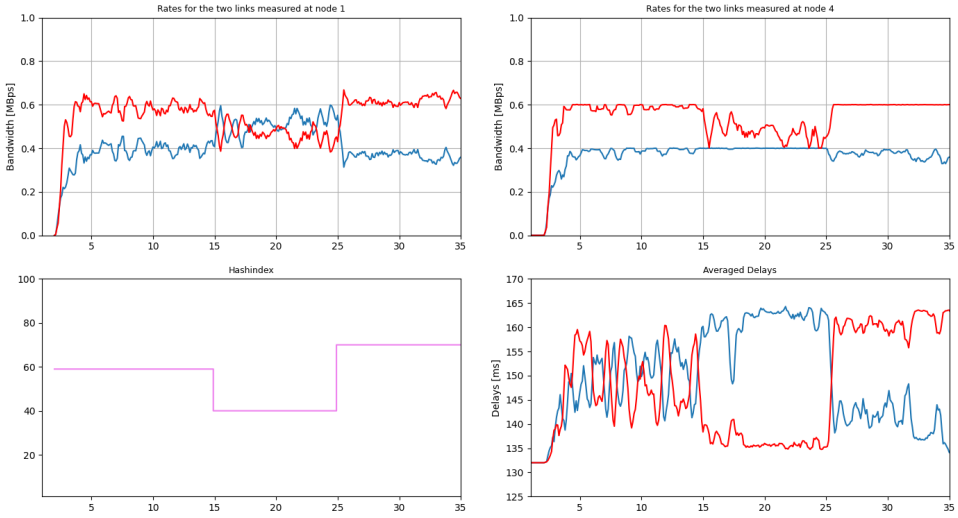

Figure 4.10: The plot shows how different hash index positions influence the process and confirms the non-linearity of the process. The two upper plots show the rate, once measured at the controlling switch and once at node 4 (congested links). The traffic rates at node 4 are saturated at 0.4 MBps respectively 0.6 MBps as these are the link limits. While the hash index is at 60, both links experience similar delay, because it is the optimal value. If it is another value as 40 or 70 the delays approach the maximum and minimum value quite fast, depending on the offset from the optimal value. This confirms the non-linearity of the system

will only influence the delay measurement when a SYN ACK of a flow in this hash range is received. If the average flow inter-arrival time is t_i (could also be the inter-arrival time of regularly sent ping messages) and the hash index change of Δh in a hash array of length l is applied, the system's dead time is:

$$\mathbb{E}[\text{Dead Time}] = t_i \frac{l}{\Delta h} + RTT \tag{4.13}$$

The RTT may change over time as queues can have different occupancy, resulting in different queue delays. Next to that, the system is very noisy. Due to the burstiness of traffic, certain packets experience almost no queue delays, whereas others wait for the whole queue to empty.

4.3.3 Tuning

The tuning rules for this process are just broad guidelines and should indicate issues to take care of. For scaling we need the maximum and the minimum delay. Opposed to the maximum traffic rate, which is known to the operator, the maximum and minimum delay can have an arbitrary value that can change over time. In the simulation, we know those values, but for a real-life scenario, one has to fix a value based on historical data or have it updated regularly. Setting it too high will just result in a slower controller.

W and T can be set according to Section 3.1. W has to be big enough to arrive at a stable signal, but not too big as it would decrease the speed of the system.

When choosing K_p one has to take into consideration, that even when the hash index is at the optimal value, the error signal can still fluctuate, resulting in an unnecessary hash index change. Suppose, an operator has a system with a hash array of length l, where the error signal fluctuates in the range Δe_{opt} and he wants to prevent that the hash index is changed more than Δh during such fluctuations. For such a requirement K_p must be chosen as the following:

$$K_p \leq \frac{1}{\Delta e_{opt}} \frac{\Delta h}{l} \tag{4.14}$$

So in our case, $e(t)$ can have values up to 0.2, even for the optimal input. We do not want any unnecessary hash index changes to happen, so K_p should be smaller than 0.05:

$$K_p \leq \frac{1}{0.2} \frac{1}{100} = 0.05 \tag{4.15}$$

K_p is fixed now and a suitable value for τ_I is needed. It is responsible for the time it takes until steady-state offset is corrected and may also react to noise if it is too high. Normally, the integrator part depends the least on noise. But because this system has a lot of noise, the proportional part is chosen so small that to drive the system to the steady-state in a reasonable time, the integrator gain must be high and gets therefore susceptible to noise. So it is a trade-off between a stable hash index and the speed until the optimal input is reached. On top of that, if the integral gain is too high for a slow process, it may accumulate error and overshoot, resulting in oscillation. A process is slow if the minimum delay of a packet is high or a large W is used. In Figure 4.11 controllers with $\tau_I = 1, 0.5, 0.25$ in a scenario where one link has 0.3 MBps and the other 0.7 MBps are shown. For $\tau_I = 1$ (first row in the plot), the controller is very stable once it reaches the optimal value, but it takes very long to get there. The contrary matches the third controller with $\tau_I = 0.25$. For our case, a good trade-off is achieved by setting $\tau_I = 0.5$, but an operator can choose based on his requirements.

So when K_p is fixed, it is best to start with a big τ_I (small integral gain) and increasing it until noise has an impact on the hash index or the system overshoots and starts oscillating after incidents.

4.3.4 Controller in Action

The controller is put to test, by having two equal size links (0.6 MBps), but an additional sender is connected to node 2, sending traffic to the receiver. The additional sender has bursty traffic patterns, making it hard for the controller. Sender 1 sends around 1 MBps on average, whereas sender 2 starts flows of total 0.5 MBps for 5 seconds at 8s and 28s. The controller tries to equalize link delays. During the first traffic burst, it moves the hash index to around 70, resulting in congestion in both the links as there is not enough capacity. Once sender 2 stops sending, the index moves down to at least 60 in order not to congest one of the links. One may think it should move back to 50, but if the goal is purely having equal delays, any value between 40 and 60 is fine as no link is congested.

4.4 Controlling Loss Rate

In this Section, the controller distributes the traffic based on the loss rate. Loss is also a measure for congestion or just faulty links which should be avoided when making forwarding decisions. It can be measured by looking at TCP re-transmissions, as they indicate that a packet got lost on the way to the receiver. Sequence numbers per flow are monitored, to register re-transmission. One

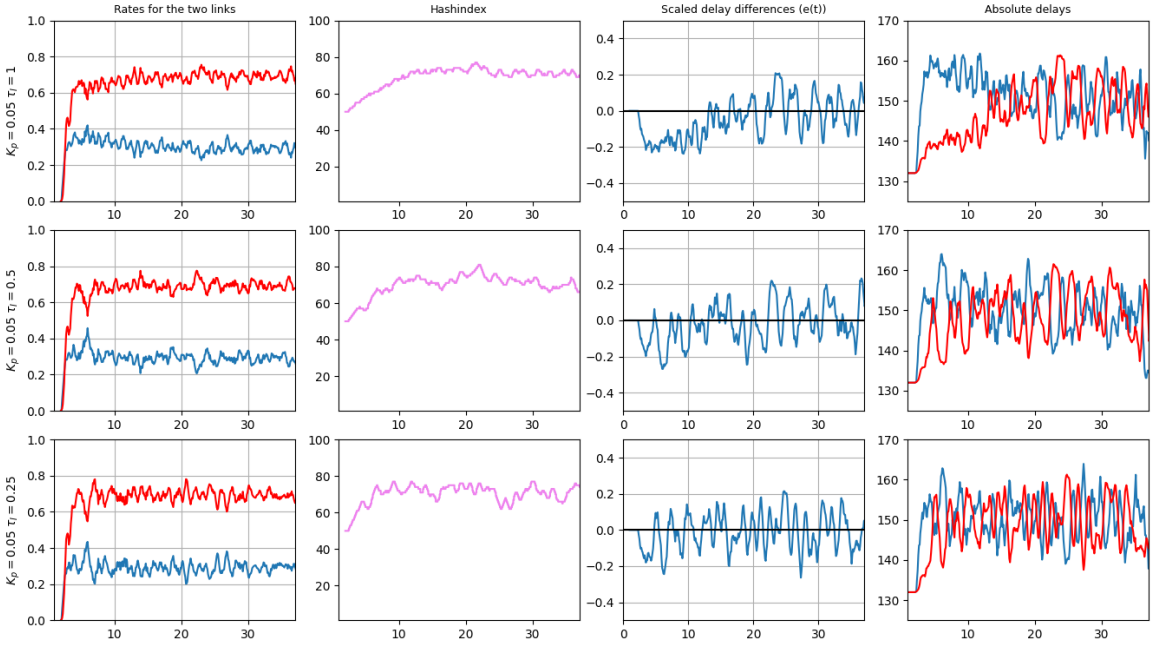

Figure 4.11: Controllers with different integral gain. Each row corresponds to one controller, showing that a small integral gain results in a slow but stable response, whereas a big integral gain has a faster response with oscillation.

way of doing that is to just store the highest sequence number seen for a flow and if a packet with a lower number arrives, a loss must have happened. The loss is registered for this link and the sequence number is updated to this lower number of the re-transmitted packet. This technique has some flaws in a real network, but in our case, it is enough. One inaccuracy would be, that re-transmission is always interpreted as a loss in the forwarding link for that flow. But the loss could have also happened earlier in the network, where we have no control of. In this case, the controller should not count it as a loss. Because in our topology, there is no loss possible before the controlling node, the measurement is accurate.

In a setting, where a flow has always the same forwarding link, this would be the only issue, but now when flows may switch the link, another issue occurs. When a flow has switched links and a re-transmission happens for a sequence number, for which both links have seen higher ones already, we do not know on which link this loss has happened by just storing the highest number seen. By storing more than that, this problem could be solved but in this thesis, we neglect this issue, as this rarely happens.

Loss can either be registered as just one lost packet or the bytes of the re-transmitted packet. We chose the first option.

The number of lost packets must be scaled in a way to get a signal between 0 and 1. This can be done by just dividing by the number of packets sent during W. Dividing by this value would get a signal, where most of the measurements are in the lower part of the range from 0 to 1, as it is more likely that some of the packets get lost compared to all of them. Additionally, as an operator, 20 % lost packets could already be considered as the worst-case. So he does not differentiate between

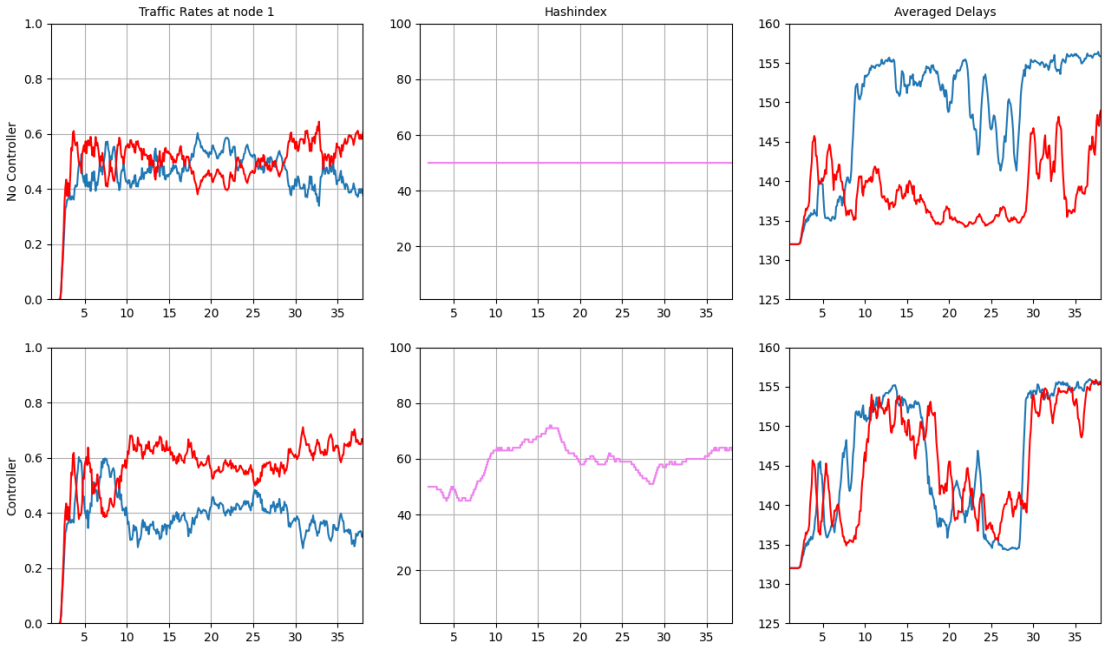

Figure 4.12: Traffic is sent over the two links and at 8s and 28s, an additional sender at node 2 starts sending traffic. This congests one link. The controller moves some initial traffic from one link to the other link, by increasing the hash index, as there is still some capacity left, while ECMP is not fully utilizing the bandwidth of the non-congested link.

20 % loss, 50 % loss or even 100 % loss. This worst case loss ratio will be called l_{max} (0.2 for 20% worst-case loss). Therefore, the number of lost packets is divided by the number of sent packets, that are considered the worst-case and if even more packets are dropped, the signal will stay at 1 (Equation 4.16).

$$l_i(t) = \frac{\#\text{lost packets}}{l_{max} \cdot \#\text{sent packets}} \tag{4.16}$$

$$y_i(t) = \begin{cases} l_i(t) & \text{if } l_i(t) < 1 \\ 1 & \text{else} \end{cases} \tag{4.17}$$

In the experiments only queue loss is simulated and there is no link loss.

4.4.1 System Identification

The dynamics of the system with loss rate as process variable are similar to the delay system. It is also non-linear because if the hash index is a bit off the optimal value, links will get fully congested over time, independent of the hash index. So the number of lost packets is not proportional to the hash index. Consequently, general tuning rules can not be applied.

Nevertheless, we show that the system has quite a slow response and a big dead time. Changing the hash index, will influence the loss rate immediately, but because our signal measures the re-

transmission, the influence of the input to the output will not be seen until packets, sent after the change, are re-transmitted. This takes about the RTT of the packet because TCP will resend when an RTO (re-transmission timeout) happens. This is even longer than for the delay measurement which was just the time from the controlling node to the sender node and back, whereas here it is the total RTT of the packet.

We neglect the time it takes until a packet is sent in the changed hash index interval Δh, as it is much shorter.

$$E[\text{Dead Time}] = RTO \tag{4.18}$$

When scaling by the number of packets sent during W there is an issue, that has an impact on the dynamics. For the measurement, we count the number of packets and the re-transmission during $[t - W, t]$, but the packet loss may have happened way earlier where the traffic rate was different. So when the signal is scaled by the current number of packets the loss rate is not precise.

The consequences can be seen in an example. Let us assume many packets got lost during $[t-W, t]$, where the traffic rate was high. This loss is measured at first during $[t+RTO-W, t+RTO]$, because we measure the re-transmissions. If during this time much fewer packets are sent, the amount of lost packets will be scaled by a smaller number resulting in a huge loss rate. Due to that, the error signal may be much higher than reality and the controller applies to much input, which may result in an overshoot. A countermeasure would be, to just take earlier packet rates as the scaling factor. For simplicity, we still just divide by the current packet count, as this inaccuracy only results in a chance to overshoot, which can be prevented with smaller gains.

This measurement is not as noisy as the delay, because all the packets are monitored and not just SYN and SYN ACK.

4.4.2 Tuning

Tuning is done similarly as in Section 4.3, especially for W and T, but also for K_p (Equation 4.14). After finding a suitable K_p, that does not react too much to noise, τ_I is chosen to give a reasonable response where the steady-state is reached fast, without overshooting. As the system is a slow process, the integral gain should be small, because the controller would overshoot if it is too high.

Different integral gains are evaluated for a K_p of 0.05, confirming that a too large integral gain results in a overshoot followed by oscillation (see Figure 4.13). It can also be seen that a slow controller is not necessarily bad, it just takes time to reach the steady-state. So if one needs to find the integral gain parameter for his network, it is better to start very low and increase it, if no overshoot happens.

4.4.3 Controller in Action

Similar to Section 4.3.4, the two links have equal bandwidth (0.7 MBps) for the test scenario, but an additional sender is connected to node 2, sending traffic to the receiver (see Figure 4.14). Sender 1 sends around 1 MBps on average until 20s, where the rate is increased to 1.2 MBps, whereas sender 2 starts flows of a total of 0.5 MBps for 5 seconds at 8s. At the beginning, where only sender 1 sends traffic, the same number of packets is lost, but as soon as sender 2 starts, the link from node 2 experiences more loss and after the controller has registered it, the hash index is moved to roughly 70 to have both links equally congested. After sender 2 is finished and sender 1 increases its rate, the hash index will slowly go back to 60 as this will result in equal loss rate. There may be more absolute packets lost in one link during certain time intervals but the goal is to balance

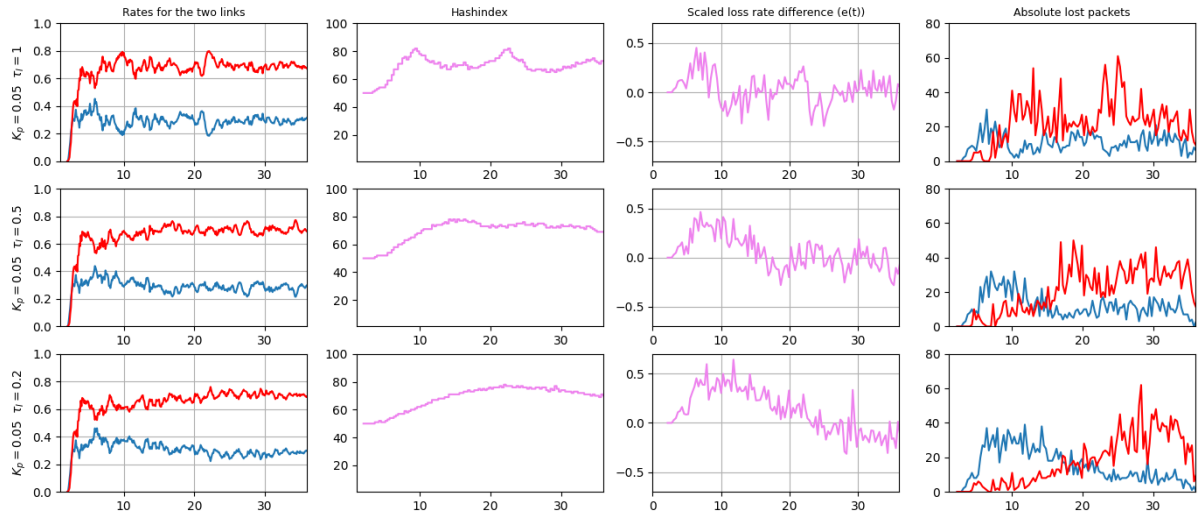

Figure 4.13: Different integral gains in a system with links of 0.3 MBps and 0.7 MBps bandwidth result in different responses. The higher we set the integral gain, the faster is the system, but it also results in a unstable behaviour.

loss rate, so this is fine. Without a controller, the link with additional traffic gets very congested, whereas the other would have free bandwidth to use.

4.5 Further Scenarios

4.5.1 More than two Links

All those scenarios were limited to the topology presented in Section 4.1.2. How do these ideas work out in the case of three links (see Figure 4.15)? For this case two hash indices h_1 and h_2 are used. The forwarding rules (Equation 3.7) and the calculation of the hash indices based on the controller output (Equation 3.8 and 3.9), can be taken from Section 3.3.

The same system identification and tuning applies to 3 links, as the signal is still in the range from 0 to 1 and the dynamics do not change. To verify this, a system with 3 links and delay as measurement is simulated (see Figure 4.16). In this scenario, the links have bandwidth 0.15 MBps, 0.35 MBps and 0.5 MBps. At the beginning the controller assumes equal distribution, so for the case with a hash array length of 100, h_1 is 33 and h_2 is 66. In the case without a controller, the smallest link gets fully congested, the biggest one has almost no queue delay and the middle one experience delays in between. The controller moves the hash index to distribute the traffic, such that the delays are equalized. This distribution matches the available bandwidth of the links.

A more difficult scenario is created by having 3 equal size links (0.4 MBps), where the sender attached to node 1 sends 1 MBps, resulting in almost no queue delay. At node 6, an additional sender sends about 0.2 MBps at 12s. As this congests link 3, it is better to move some traffic from sender 1 to link 1 and 2. By sending about 0.4 MBps over link 1 and 2 and 0.2 MBps over link 3, all the links experience a similar amount of delay. Without the controller, link 3 gets fully congested as the traffic exceeds its bandwidth.

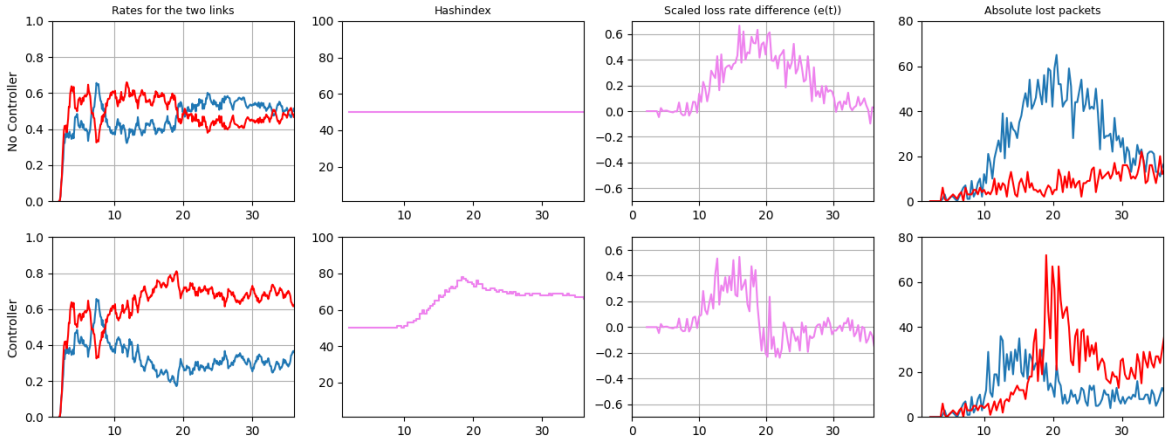

Figure 4.14: Test case scenario for the loss controller, where additional traffic is sent over one link and congesting it. The controller reacts by moving traffic to the other link, which still has empty bandwidth. With ECMP (no controller) one link gets congested and many packets are lost on it.

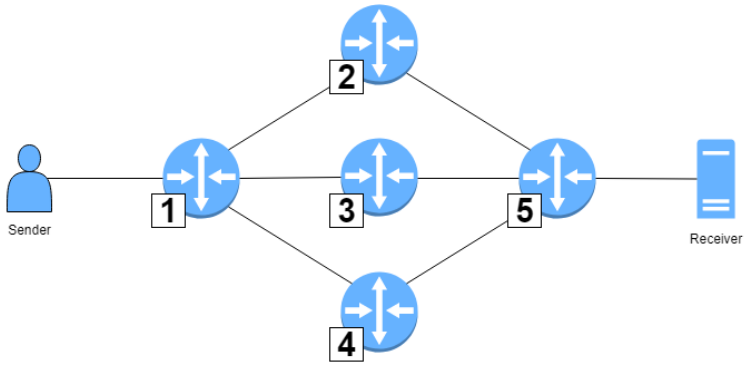

Figure 4.15: Topology with three possible links

4.5.2 Multiple Network Measures

In this Section we show that a controller with multiple measurements is beneficial. The delay system we use (Section 4.3), does not consider lost packets, which is actually a strong indicator of an imbalance or failure. Therefore, a controller is proposed, which utilizes both the delay measurement and the lost packet measurement, to get both indicators of congestion. The signal is generated according to Section 3.3.2. Both measures are weighted equally ($w_{loss} = 0.5$ and $w_{delay} = 0.5$). As the delay signal has a faster response than the loss signal and the loss signal is less noisy, they can provide a better network signal together. Controllers, which use only the loss measurement ($w_{loss} = 1$), only the delay measurement ($w_{delay} = 1$) and both of them weighted equally, are compared against each other (see Figure 4.18). The combined model removes some of the overshoot from the loss measurement and smoothes out the noisy delay measurement. This is just an example and one could combine these, or other network measures in any way.

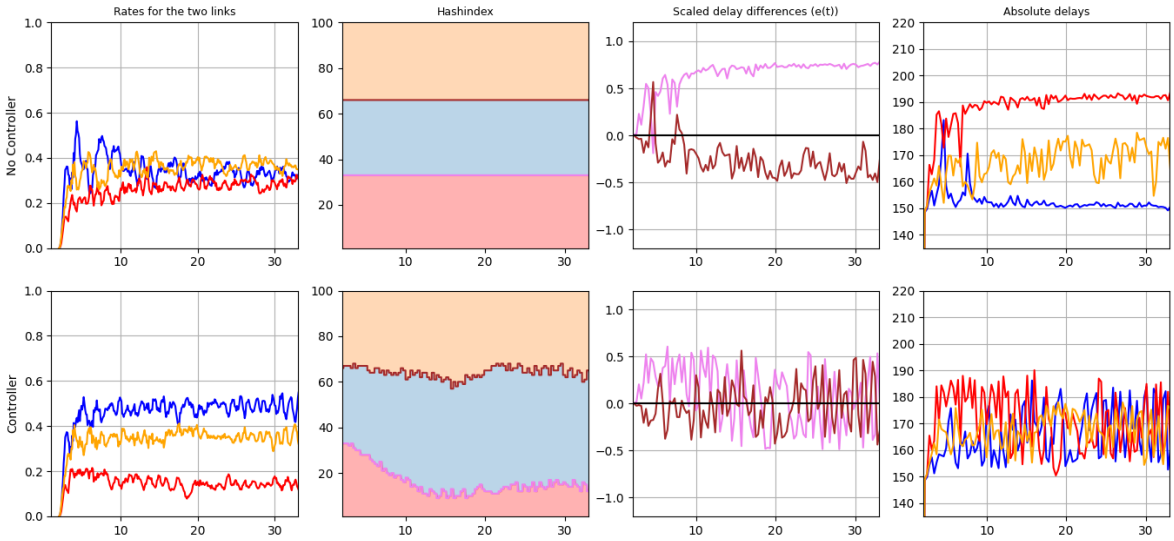

Figure 4.16: Traffic is sent over 3 links and two controllers, with their respective hash index h_1 and h_2, are used. The colors in the hash index plot represent the available hash index range for each link. The bandwidths of the three links are 0.15 MBps, 0.35 MBps and 0.5 MBps, so the controller distributes the traffic with this ratio, to avoid long delays and therefore congestion. Having no controller results in a full queues as the small links can not handle a third of the traffic.

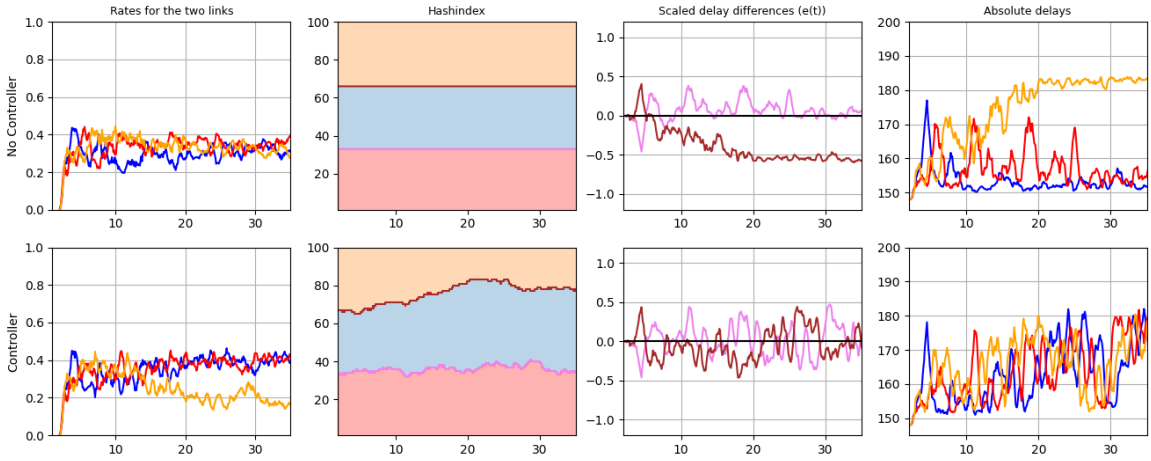

Figure 4.17: Traffic is sent over 3 links and two controllers, with their respective hash index h_1 and h_2, are used. The colours in the hash index plot represent the available hash index range for each link. The bandwidths of the three links are 0.4 MBps. Sender 1 at node 1 sends 1 MBps and at 12s Sender 2 at node 6 starts sending 0.2 MBps, congesting its link. In the beginning, the controller distributes equally but after the disturbance, it moves traffic, to avoid long delays and therefore congestion. ECMP works fine as long as there is no congestion.

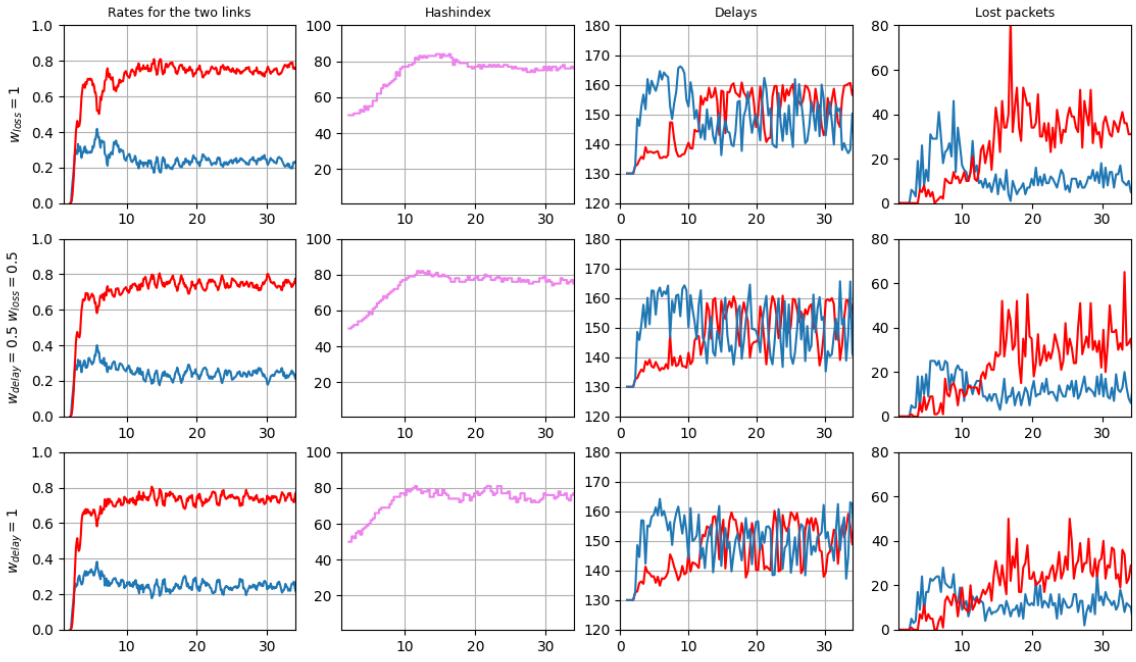

Figure 4.18: The three controllers (loss rate, combined, delay) are compared by having them adjusting the forwarding distribution to a bandwidth imbalance of 0.25 MBps and 0.75 MBps. The combined controller removes some of the overshoot from the loss measurement and smoothes out the noisy delay measurement, resulting in a superior controller.

Chapter 5

Summary and Outlook

In this thesis, we present a novel way of using network measurements for changing the forwarding behaviour with a PID controller. We further show how different network variables have completely different dynamics and therefore, a universal controller does not exist.

At first, the general controller design based on general network measurements was presented. The controller influences the forwarding distribution along the links by mapping flows to a hash array, for which the values are divided between the forwarding links. The main goal was to find possible network variables, which may be beneficial to control (traffic rate, delay and loss) and identify their dynamics by potentially model them as a first order plus delay system. If possible, one can apply tuning rules that have been proven to control the process variable reliably.

The system with traffic rate as process variable indeed behaves like a first order system. The tuning rules can therefore be applied to this system. Nevertheless, the tuning rules themselves have a tuning parameter to choose, for which there is not a general value that fits all scenarios. So in the end it comes down to trying out different controller parameters and selecting the appropriate one. Yet, we are still able to provide some guidelines on how to go about finding the right parameters.

The other two systems, which are distributing traffic to minimize congestion by measuring the loss rate and delay, are not even linear, so it is very hard to specify universal controller parameters. For easier integration in new scenarios, we still propose some guidelines, that take the network setting into account.

Lastly, we showed that multiple measurements can be combined to arrive at an even bigger range of possible controllers. As is often the case, there are trade-offs to make. One could prefer a fast adaptation to disturbances, which may include many unnecessary forwarding changes and the chance to overshoot. The other may be satisfied with a slow response but wants as less unnecessary forwarding changes as possible. This customizability is an advantage of using a parametric controller. Measuring the network variables with a procedure other than the moving average window may additionally improve the controller. Other filter possibilities like the exponential moving average window, which indicates that it does not slow down the system as much, need to be investigated further.

This thesis tested the controller in a very restricted environment. To fully confirm its robustness and stability, many more scenarios and edge cases would need to be evaluated. It may also be important to study the case where multiple nodes use a controllers, as they might fight for a link's capacity. Are they starting to oscillate or is the more aggressively tuned controller going to win the bandwidth? Is the controller selfish and just improves performance for himself and brings the entire network out of balance? All of these questions remain open.

Another topic that was only slightly touched is using a controller for multiple links. Two

controllers working together for three links seems to work well, but how can this be done for more than three links. A possible approach for four links would be to aggregate two links together and have a parent controller with its own hash array balancing out the network variable between the two pairs. Additionally, there could be two child controllers doing their job on the links that got paired together. This idea could be expanded to even more links.

In this thesis, we could not propose a ready-to-use controller for all possible scenarios, but we still came up with a system, that may not be straightforward to tune, but is able to control a variable reliably without communication between switches. Our thesis also reveals that the network's unpredictable behaviour and its non-linearity are the reasons why a ready-to-use controller is not possible and why there might be not many other approaches, trying to incorporate control theory in networks.

Bibliography

[1] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, V. T., Matus, F., Pan, R., Yadav, N., et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), pp. 503–514.

[2] Åström, K. J., and Hägglund, T. *PID controllers: theory, design, and tuning*, vol. 2. Instrument society of America Research Triangle Park, NC, 1995.

[3] Chandrasekaran, B. Survey of network traffic models. *Waschington University in St. Louis CSE 567* (2009).

[4] Holterbach, T., Molero, E. C., Apostolaki, M., Dainotti, A., Vissicchio, S., and Vanbever, L. Blink: Fast connectivity recovery entirely in the data plane. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 161–176.

[5] Skogestad, S. Probably the best simple pid tuning rules in the world. In *AIChE Annual Meeting, Reno, Nevada* (2001), vol. 77.