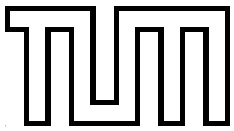


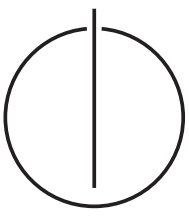
Technische Universität München

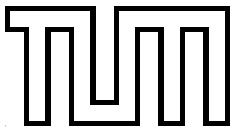
Faculty of Informatics

Bachelor's Thesis in Informatics

Congestion Control in P2P Networks with Parallel Paths

Martin Raiber





Technische Universität München

Faculty of Informatics

Bachelor's Thesis in Informatics

Congestion Control in P2P Networks with Parallel Paths

Staukontrolle in P2P-Netzen mit parallelen Pfaden

Author:	Martin Raiber
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisors:	Dr. Heiko Niedermayer Dr. Nils Kammenhuber
Submission Date:	September 20th 2010

Statement

I assure the single handed composition of this bachelors thesis is only supported by declared resources.

Munich, September 20th 2010

.....
(Signature of candidate)

Abstract

This thesis tries to find good approaches for congestion control when there are numerous paths available between two peers. The high number of paths makes individual path level congestion control impractical, as not enough congestion feedback would be available per path. Two applications where having such a congestion controller would be useful are examined: In anonymous information sharing, sending a packet over other peers guarantees anonymity, while using multiple such paths diminishes the chances of an attack where identities are disclosed. Details on how such an approach to congestion control can be incorporated into an anonymous information sharing overlay are provided. For the other application, live video streaming, an example program is developed, which uses the congestion control approaches described in this thesis to efficiently distribute a live video stream using multiple tree structures per stream. Two distributed congestion control approaches are developed. One is an adaptive increase multiplicative decrease (AIMD) approach, which is simple and lightweight. The other is a congestion controller that learns the specific client rates by the reinforcement learning approach Q-Learning. This approach uses more memory and reacts slower to congestion, but as it uses past experience to control the rates it obtains a better equilibrium throughput performance. For comparison reasons a non-distributed congestion controller is developed which has instant knowledge from all peers and therefore should perform as good as possible. Those three approaches are compared with each other and with doing no congestion control at all and just using random peers to form a path. The parameters of the different approaches are optimized with regard to as many influences as possible using the simulation environment Oversim/OmnetPP. Then the different approaches are compared. The Q-Learning approach reached the performance of the non-distributed solution in several scenarios. The AIMD congestion controller is worse than the Q-Learning approach if the number of participating peers is low and the bandwidths these peers allow the overlay to use or the latencies between peers is relatively high. With an increasing number of peers and smaller available bandwidth the AIMD approach becomes attractive, as it performs only slightly worse and uses less memory. Doing no congestion control causes very low throughput and should be avoided. It is tested how the congestion control algorithm reacts to rapid changes in available peer bandwidths. Here the AIMD approach is better, whereas the Q-Learning approach needs a lot of time to react to these changes and reach an efficient equilibrium again.

Kurzfassung

Diese Arbeit versucht gute Ansätze für Staukontrolle zu finden, falls viele mögliche Pfade zwischen zwei Teilnehmern eines Overlay-Netzwerkes existieren. Diese große Anzahl an möglichen Pfaden macht Staukontrolle auf einzelnen Pfaden unmöglich, da zu wenige Informationen pro Pfad zur Verfügung stehen. Zwei Anwendungen solch eines Staukontrollalgorithmus werden vorgestellt: In anonymen Netzen verhindert das Schicken von Paketen über mehrere Teilnehmer, dass die Identitäten des Empfängers und des Senders aufgedeckt werden können. Das Verwenden eines zufälligen Pfades pro Paket verringert die Anfälligkeit für spezielle Angriffe, durch die Identitäten aufgedeckt werden können. Es wird beschrieben, wie eine Staukontrolle in solch ein anonymes Netz integriert werden kann, um die Geschwindigkeit zu verbessern. Für die andere Anwendung, Echtzeitvideostreaming, wurde ein Beispielprogramm entwickelt, das die in dieser Arbeit entwickelten Staukontrollalgorithmen verwendet um effizient einen Videodatenstrom zu verteilen. Zwei dieser verteilt agierenden Staukontrollalgorithmen werden entwickelt. Einer ist eine Adaptive Increase Multiplicative Decrease (AIMD) Variante, die relativ einfach und speichereffizient ist. Die andere Variante ist eine Staukontrolle, die versucht verschiedene Teilnehmerraten mithilfe eines Reinforcement Learning Algorithmus (Q-Learning) zu lernen. Das benötigt mehr Speicher und die Staukontrolle reagiert langsamer auf auftretende Staus, aber die erreichten Geschwindigkeiten sind höher. Um diese zwei Ansätze einordnen zu können wird noch ein unverteilter Algorithmus entwickelt, der theoretisch bestmögliche Staukontrolle durchführen sollte. Diese drei Ansätze werden dann miteinander und mit dem Fall verglichen, dass keine Staukontrolle verwendet wird. Die Parameter der verschiedenen Algorithmen werden mit der Hilfe von Simulationen in Oversim/OmnetPP optimiert. Anschließend werden sie verglichen. Der Ansatz mit Q-Learning erreichte in den meisten Szenarien die beste Geschwindigkeit der verteilten Algorithmen. Der Ansatz mit AIMD ist schlechter als der mit Q-Learning, falls die Anzahl der Teilnehmer gering und deren Bandbreiten oder deren Latenz zueinander relativ hoch ist. Mit einer steigenden Anzahl an Teilnehmern und kleineren dem Overlay zur Verfügung gestellten Bandbreiten wird der Ansatz mit AIMD zunehmend attraktiv, weil er dann nur geringfügig schlechter, aber viel speichereffizienter ist. Wenn keine Staukontrolle verwendet wird, sind die Ergebnisse sehr schlecht – dies sollte also vermieden werden. Es wird außerdem getestet, wie die Ansätze auf sich schnell ändernde Teilnehmerbandbreiten reagieren. In diesem Fall ist der Ansatz mit AIMD besser, weil der mit Q-Learning eine längere Zeit braucht, um sich auf die neue Begebenheiten einzustellen.

Contents

Chapter 1.
Introduction

Overlay networks are networks build on top of other networks. If this overlay network consists of equally privileged nodes one speaks of a peer-to-peer overlay network. Those are widely used in the Internet [1]. Large data quantities are sent over these networks. Since network resources are always limited, it is important to use the available resources as efficiently as possible. Resource limitations are sometimes found in the provider networks but are more often found at the peers and their connection to the network of the provider – the so called last mile.
We often have multiple possibilities to obtain the data we want. In BitTorrent[2] data is distributed as much as possible in a group of peers (swarm) so that many possible upload and download targets exist. To use available resources efficiently, targets have to be selected that have the free capacities to receive what one sends them.
Using all available paths at once avoids the problem of having to select and probe for a few good paths. The problem becomes more easily manageable as the probing and selections can be done implicitly by a congestion control algorithm which distributes the load between the different paths available. The problem degenerates to a simpler estimation of individual optimal peer rates. Two different approaches to control these rates are developed and tested in this thesis. The applications for these controllers are widespread. Overlay networks for anonymous information sharing have to send data over multiple participating peers to hide the sender's IP address. If the overlay allows sending the information over multiple paths such a congestion controller can be used to more efficiently use available resources.
In live video streaming single video data packets sometimes have multiple available paths with varying efficiency, with regard to the performance (latency of video stream and sustainability of clients). The multiple path congestion control can be used to choose the optimal one.
It is even possible that sending data via another peer is more efficient than to send it directly to the receiver. Such situations could be recognized by the congestion controller and exploited.
This thesis will look at the abstract problem to develop and test various congestion controllers. The congestion controllers will be able to decide which one of the numerous amounts of paths available will result in the overall best resource usage. Both a distributed and non-distributed solution of the problem is devised and compared. Several simulations are done to optimize each congestion controller. Afterwards they are compared in different scenarios and the results are interpreted.

1.1. Congestion and the Internet

Anyone who has ever driven a car before knows that congestion can be a huge problem in car travel. People are trying to travel to the same places at the same times. Traffic systems sometimes do not have the capacity for the amount of travelers and become congested. This in itself is not a problem; the problem is that congestion at one place can cause the traffic system of a whole city to fail as an accumulation of cars clog the streets and nothing moves. Here the analogy of real and Internet traffic ends, because the cars in real traffic are humans who (sometimes) know beforehand of such situations and avoid driving over congested streets at times they know will be congested. In contrast, Internet packets are dumb. They cannot select their route and are just sent around and there is often only one path available between sender and receiver at one point in time – this is the route the routers on that path deem the best for the source-destination pair. Another discrepancy is the way routers react to congestion; if their buffers are full, additional incoming packets are discarded, whereas in street traffic the congestion just continues to build. In a way, a real collapse where absolutely nothing goes anymore is unlikely – after the router works through packets it accepts packets again. The worst case would be if sources detecting congestion would increase their sending rate in the hope that at least some packets get through. Then the buffer would fill even faster and other connections would be severely disrupted. If the users just continue sending at the same rate, occasionally packets will get lost and users with higher rates will get more data through (hence the selfish idea of increasing the rate). This is, however, unfair and should be avoided. Clients who want to get data from A to B reliably especially have huge issues with dropped packets as they have to wait for at least the estimated round trip time and then send the packet again. This means that their actual rate, with which data is sent, decreases. This is where we need a congestion control algorithm, which increases rates if there are resources free and decreases them if a router on the path is congested. If everyone uses a congestion controller, everyone wins – with winning meaning getting higher throughput. Since situations in which the routers overflow are avoided, sources do not have to resend packets and can use all of their available bandwidth for usable (not duplicate) data. Note, however, that this is not a Nash equilibrium (situation in a game where one player does not gain anything by changing his strategy), because if one source decides to change the strategy and sends more if it detects congestion and all others avoid congestion, the one changing the strategy clearly wins (has more throughput) – all other sources decrease their rate. Such behavior is luckily not seen very often.
As we have seen, everyone gains by employing a congestion control algorithm and therefore one is heavily in use in the Internet: It is part of TCP that has been around since the early days of the Internet, has had made its mistakes and has been improved (see [3]) and today it is responsible for what we consider a fast and stable Internet.

1.2. Problem

Suppose that node[1] A does not want to send packets to B directly. There are a number of possible reasons for that:

1. Privacy issues
   A does not want B to know which address A really has. To reach that goal, A has

[1] Here used as equivalent to peer

to construct a path with willing relay nodes in it. A then sends the packet to a relay node which sends it to other relay nodes or to B. Now B only knows the address of one of the relay nodes and not of A. B has usually some identifier which does not disclose the address with which A can contact B without knowing the address of B.

2. Throughput improvement
   The connection from A to B is slower than from A to B over a relay node. It is better not to directly send data. Or it could be the case that A can increase the throughput to B by additionally using a connection over a relay.

3. Structural requirements
   A is in some kind of structured network where it can contact B only indirectly via other nodes. This is, for example, the case in distributed hashtables or if B is behind a firewall that allows A to contract C and C to contact B, but not A to contract B directly.

Because A does not want to send packets to B directly it has to select the relay nodes for a path to B (see Figure 1.1). If we have a decent amount of willing relay nodes, we have a huge amount of available paths. For example, with 50 nodes available and 3 hops between source and target, we have $50 * 49 * 48 = 117600 (= \frac{n!}{(n-h)!}$ with $n = 50$ and $h = 3$) different paths available.
The users will usually not want to spend their whole bandwidth on relaying traffic, but rather will want to set a hard limit on relay data, such that enough bandwidth is available for other more directly useful kinds of traffic. The only option relay nodes have regarding unwanted traffic is discarding it and hoping that fewer users decide to send data using them. This is exactly what a congestion control algorithm should do if it detects lost packets. If we select only random nodes for the paths such over usage can occur and disturb other applications and data streams on the overloaded relay node.
Due to the large number of paths it is not feasible that the senders do congestion control on an individual path level. If we want to do individual path level congestion control it must be performed for a selection of paths, otherwise we would get too few congestion feedback signals because the probability a path gets selected is very low. We would have to take a subset of the available paths, look at how well they perform and continuously sample new ones. This can lead to oscillations. More importantly, if we use the multiple paths because of privacy issues we give the participants of the subset more information about ourselves, as we send more information over them.
If we assume the congestion occurs at the periphery of the network, we can use all paths and do the congestion control on the nodes themselves, as this is where the bottlenecks are now. We have better privacy and the oscillations decrease as we now have a larger number of paths available.

We are now ready to summarize the problem:

Input

- Node A, which wants to send something to another node B using h other nodes as intermediate hops.

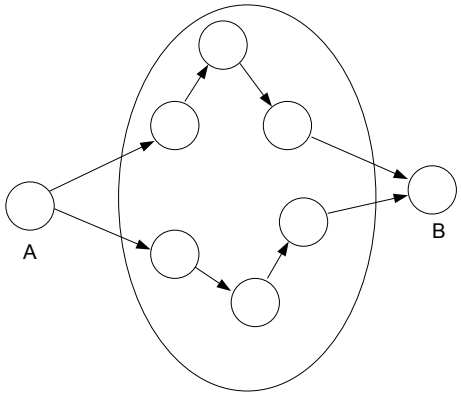


Figure 1.1.: A wants to send something to B using multiple hops. It has 6 different nodes available for building the path. Here are 2 possible paths with three hops.

- n other nodes which are known to A offer bandwidth for relaying packets, but may also be used by other nodes as relays. Each of these nodes N_k limits the packets it is willing to forward by some value R_k.

Output

One of the $\frac{n!}{(n-h)!}$ paths available to A that aims to respects the forwarding constraints R_k of the nodes on that path.

Limitations

- A does not know the R_k or which/how many other nodes use N_k as relay.
- We can only accept a distributed, opportunistic approach, where each sending node tries to maximize its own throughput. By doing that the throughput of the whole system gets maximized.
- Congestion is assumed to occur at the periphery of the network (see Figure 1.2). That means if we have nodes C, D and F; the connection from C to F is congested and the connection from C to D is not, it follows that the connection from D to F is congested as well – the periphery of C is not congested, because C can send to D. It follows that the periphery at F is congested. If this assumption does not hold, all techniques shown in this thesis will yield lower performance than demonstrated. Then a packet loss does not always indicate that a node is overloaded, but can also mean that the path from the previous relay node to the next one is congested. Other relay nodes then may not share the bottleneck where the congestion occurred and thereby would have been able to send the packet without congestion.

1.3. Motivation

We have already mentioned the motivation for using multiple parallel paths: Privacy issues, structural requirements or throughput improvement. Here we will look at two applications to motivate multiple parallel paths further. One is anonymous information sharing, which clearly falls under 'privacy issues', the other is streaming which falls under 'structural requirement', because if the bandwidth of the streaming server is limited and we do not send messages over several nodes less peers would be able to receive the stream.

1.3.1. Anonymous information sharing

People want to share information without being identified. The most recent popular application for that is WikiLeaks (http://wikileaks.org) where people can leak information anonymously. It uses TOR[4] as an anonymous overlay network to disguise the IP addresses of the informants that publish information. TOR uses Onion routing[5] to establish anonymous routes from sender to receiver. Using a low number of routes exposes the sender to an arsenal of timing and pattern attacks. If we use all available relays for a huge amount of different routes, these kinds of attacks become ineffective. This is done in another approach to build an anonymous overlay network – MORE[6]. There each packet takes a different route.
Obviously we now have exactly the kind of congestion control problem this thesis is about. If we find a good solution to that kind of problem we can apply it to MORE and thereby increase the throughput in that anonymous peer-2-peer network.

1.3.2. Streaming

We have to distinguish different kinds of streaming: End-to-end, one-to-many and peer-to-peer-one-to-many streaming. We can further separate them into live and delay insensitive streams.
End-to-end streams are for example voice calls via Internet. One-to-many streams are e.g. radio or television stations in the Internet. In peer-to-peer-one-to-many streaming a peer-to-peer overlay networks helps distributing the stream that one source sends.
Delay insensitive streams are, for example, a film or music stream where some delay is acceptable and a buffer can be used.
Live streams are, for example, a video and/or audio conversation or a live soccer game. We want the delay between sending the information and displaying or playing it to be as small as possible. We cannot apply congestion control: if we lose a packet because of congestion or for some other reason, we do not resend it, because that would only make the situation worse – either we have enough bandwidth available or we do not. We can, however, adjust the streaming quality, but this can often be done only in coarse steps and as such only with a high reaction time (See e.g. [7]). Streaming applications often do not care about single packet losses. They are bad for existing AIMD (See 4.1) algorithms like the one in TCP, because these act severely on even losing one single packet by multiplicatively decreasing their rate. This means they get a smaller share of the available bandwidth which is not fair.
This thesis will show how peer-2-peer congestion control with multiple paths can be applied

to delay insensitive peer-2-peer one-2-many streaming such that better performance than other state of the art approaches can be achieved. This therefore motivates the research into finding a good solution to the peer-2-peer multiple path congestion control problem, as well.

1.4. Approaches

Several approaches are devised in this thesis to solve the multiple parallel path problem. These solutions are later implemented in a simulation and compared to each other.

1.4.1. AIMD – Adaptive Increase, Multiplicative Decrease

A congestion controller that increases the rates to different nodes adaptively on no congestion and decreases the rates multiplicatively will be developed, optimized and compared to the other solutions. An AIMD-algorithm increases its sending rate linear on success and decreases it exponentially on a loss. Most of the transfers in the Internet use TCP. The congestion control in TCP is a representative of AIMD. If we use AIMD for our congestion controller as well, we will be fair to TCP – if we use the same parameters for the increase and decrease. This means that if a TCP stream and a connection to a node that is congestion controlled by one of our algorithms share a common choke point, both will get the same amount of throughput.

1.4.2. Q-Learning

AIMD is ineffective for streams with random packet losses that are not caused by congestion (e.g. caused by interferences in wireless networks), high bandwidth delay products and for inexact timeout estimations. All these things occur increasingly if we have a high number of hops from sender to receiver. The delay between congestion and congestion signal is bigger. With a larger number of hops, the probabilities for random packet loss increase rapidly as they are the multiplied probabilities for random packet loss for each hop on the route [2]. The variances of the round trip times become bigger with increasing hop count as well.
To do a better congestion control the Q-Learning congestion controller is developed – a Machine Learning approach that is able to learn the optimal rates the nodes are able to handle.

1.4.3. No congestion control

To be able to asses the possible benefits of congestion control, we also evaluate a scenario that does not employ any congestion control at all. This means we simply use random

[2] If p is the probability of random packet loss for each section on the path and we have h sections. The probability for packet loss on the whole path is $P = 1 - (1 - p)^h$. With a loss probability of $p = 5\%$ and $h = 4$ it follows $P \approx 19\%$.

nodes to build our routes. As each participating peer which sends something does this, on average all peers would be used equally – the load would be distributed among the peers.

1.5. Overview

The remaining part of this document is structured as follows: The next Chapter will present two applications, where a congestion control approach for the multiple parallel path problem can be used to improve performance.
Then in Chapter 4 and Chapter 5 two different approaches that try to do congestion control in a multiple parallel path scenario will be presented.
Chapter 6 shows how they and one of the applications (live video streaming) were implemented.
The approaches will be optimized and compared by simulation in Chapter 7.
The results and the thesis is summarized and an outlook is given in Chapter 8.
Appendix A gives two algorithms to do undistributed congestion control and several plots to justify the comparison results in Table 7.4.

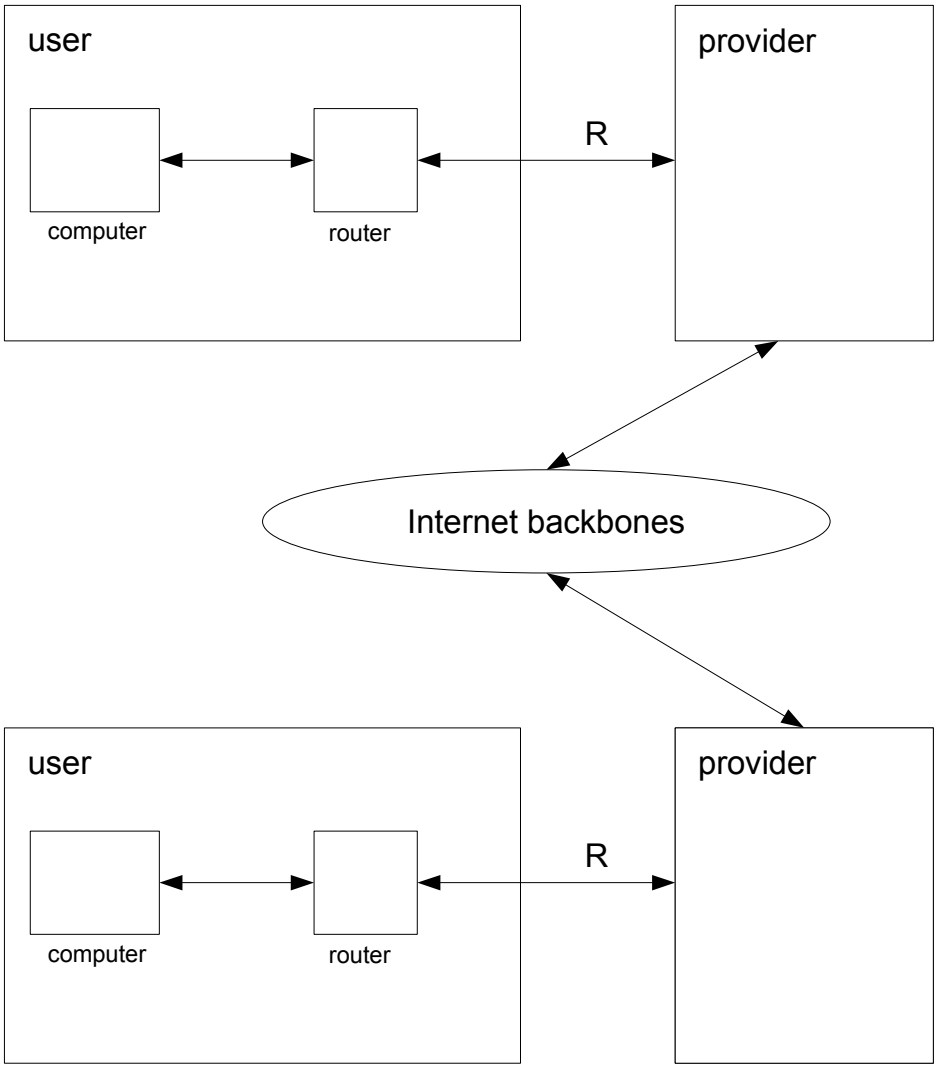


Figure 1.2.: A connection from one client to another is shown. Congestion is assumed to occur only at point R – the connection between the user's router (or computer if he is connected directly with his provider) and provider or on the clients in software (bandwidth limited by application). This is nowadays a valid assumption as the providers and Internet backbones should have some reserves.
In our applications the clients often further limit the used bandwidth to some fraction of the maximal capacity of R. This makes it even more likely that the periphery assumption will hold.

Chapter 2.
Related Work

In this chapter work related to this thesis is presented. Finding related work is difficult because most work focuses either on optimizing congestion control along a single path, or assumes that there are only a small number of paths available (e.g. about 10 paths). Sometimes new paths are sampled and bad ones are discarded. We, on the other hand, use all available paths and through the periphery assumption are able to deal with all of them, thus shipping around lots of problems that otherwise would be difficult to solve (How many, how often and how long to sample for paths; how many paths to use in the first place; how to avoid peers simultaneously selecting the same path and later discarding it – because it is used to heavily – at the same time again, causing oscillatory behavior).
Internet Service Providers (ISPs) try to use their networks as efficiently as possible. This problem is similar to the multiple parallel path problem, in the way that packets have a very high number of paths available from source to destination and different from it because it has less constraints: For a start the number of hops between source and destination is not important. In our problem we want paths with a specific number of hops. Secondly the ISPs want to optimize both throughput and sometimes latency. Given a high connectivity the network is often optimized for a minimal maximum link usage only using shortest path. We only optimize with regard to throughput. Thirdly it may be, that the assumption that the congestion occurs at node and not at link level, that we make for the multiple parallel path problem, does not hold.
To optimize their routing system the ISPs employ traffic engineering techniques. Popular are non-distributed approaches that build optimal routing strategies using (worst case) samples of past traffic, assuming that future traffic will be similar. For example one can use Open Shortest Path First(OSPF) where traffic gets equally distributed along the shortest paths. By manipulating link weights one can then find a well performing solution [8, 9, 10]. This calculation is done offline and on a central point and takes several hours and can thus react only to long term changes.
There are dynamic multipath traffic engineering protocols which can be used to efficiently route packets in ISP networks. Those try to solve the routing problem the ISPs have in a distributed manner and are thus more related to the work done in this thesis. MATE[11] is one such protocol. Other differences become apparent: For one thing there is not just a binary congestion signal indicating if there is congestion or not, but nodes can report back how congested they are and for another thing every node can be trusted to cooperate – in our approach there could be malicious peers. MATE even assumes an oracle that has global knowledge of the network. TeXCP[12] does not have such an oracle but requires congested routers to continuously send congestion feedback to avoid oscillatory behavior between the per default ten paths it optimizes the utilization for. This congestion feedback is calculated per flow using AIMD, which we will use in Chapter 4 to solve our problem.

The individual path weight is then calculated using this congestion feedback. MATE and TeXCP are given a number of existing, equally efficient paths between the source and a destination and will balance the load between those paths. Another dynamic multipath traffic engineering protocol, RePlex[13], needs those equally efficient paths as well, but router cooperation is optional. The sampling and selecting technique of the paths is loosely similar to the approach taken in Chapter 5 to find the optimal throughput for a node, but from a game theoretic view. There a large number of agents try to opportunistically improve their individual performance with the (α, β)-exploration-replication technique. They sample a new path with probability β and with probability $(1 - \beta)$ take a path chosen by its popularity among other agents. The random sampling ensures that each path is being continuously probed, the proportional sampling uses the "'crowd wisdom"' of the agents to make a better decision. The agents only migrate to a sampled path if it is better than their current one and only with a probability that depends on how much better the sampled path is than the current one. This avoids oscillations. P-STARA[14] uses the same principle to route messages in wireless networks. It is a real routing mechanism that is supposed to find optimal (here with regard to latency) outgoing links on each router in a distributed manner. Both RePlex and P-STARA assume that the path latency represents the path's congestion. Newer TCP variants use higher latency as indication for congestion as well – we cannot make this assumption as we do congestion control for nodes and latency can only be measured per path.

2.1. Multiple path AIMD

There is already work available regarding multiple paths and AIMD but most of this work tries to solve a different problem, where there are only a few number of paths available from one node to a resource. The node then selects a subset of this paths, samples them, and replaces slow ones. In this scenario [15] shows that, with a congestion controller that does not coordinate the multiple paths, AIMD with RTT bias such as TCP Reno (See section 4.1) attains only inefficient equilibriums, whereas algorithms with the RTT bias removed (TCP Vegas or CTCP[16]) can reach a fair, efficient and stable equilibrium. It also shows that a coordinated congestion controller exists that achieves the same goals, but does not explicitly present it.
Close to the above problem is the situation where we have multiple paths available from one source to a target. That could be caused by one of the nodes being multi homed (e.g. a smartphone with both wireless and carrier access). We then want to use these multiple paths to increase stability and throughput. The main idea is to extend TCP so that the multiple paths in a single TCP stream are hidden from the user applications, while still giving the profit of stability and increased rate.
Much theoretical work has been done like [17] or [18] and there are even some IETF drafts regarding this problem. See "'Draft-ford-mptcp-multiaddressed"' for general information on how TCP should be extended to use multiple paths and "'draft-raiciu-mptcp-congestion"' on how congestion control should be done.
MPTCP tries to avoid taking more capacity than a single path flow. If the multiple paths are routed through the same node doing uncoordinated congestion control would be unfair to TCP connections which only have one path, as each of the uncoordinated paths would get the same share as the single path TCP. As we do not have such restrictions in our case, this work is not applicable to our problem as well.
The approach developed in this thesis uses work done for the congestion controller in TCP

[3] to develop the corresponding multi-path congestion controller. Because it is rate based RAP[19] was used as inspiration as well. RAP is a simple, rate based AIMD congestion controller.

2.2. Reinforcement Learning for Congestion Control

In Chapter 5 we will use Reinforcement Learning[20] to do the congestion control on the multiple paths. This has not been done before. In [21] it was successfully applied to routing packets under various load situations and with different available paths.
In [22] a learning Congestion Controller is developed, but it needs an additional non-binary congestion indicator. In [23] Fuzzy reinforcement learning is used to do congestion control in high speed networks. But they do a multi-agent congestion control where information about what other controllers are doing is available. Fuzzy Reinforcement Learning was also used as a way to control the source rates based on observations made in the network and thereby avoiding congestion in [24] and [25].
The approach in this thesis uses discrete states and actions and can thereby use simple Reinforcement Learning algorithms as described in [20], which is also the main source of the algorithms and methods used for the Q-Learning congestion controller.

2.3. Anonymous information sharing

This thesis mainly looks at an anonymous overlay named MORE[6] which is based on Onion routing (see [5]) and how congestion control with multiple parallel paths can be incorporated in it. The authors of MORE had problems with congestion control and thought about using an algorithm similar to the AIMD approach developed in this thesis. Currently they seem to use no congestion control on the paths at all, which, as we will see, gives very poor results.

2.4. Video streaming

An application which allows live video streaming using the developed congestion controllers is developed. There are two competing concepts on this field. Tree based streaming where all streaming content is delivered in tree structures and mesh based streaming where a tree is used to disperse the streaming packets and peers build meshes to exchange these dispersed packets with each other in a BitTorrent[2] like fashion(See e.g. [26]). Mesh based streaming has shown better results in areas like capacity usage or susceptibility to churn[27] and is thus being favored both by research groups and applications. In this thesis a purely tree based approach is developed using concepts from [28] where the video stream is separated into different slices and for each slice an individual streaming tree is created.

Chapter 3.
3 Applications

In this chapter two applications in which the congestion controller for multiple parallel paths can be used are presented. One is a theoretical description of how such a congestion controller could be incorporated into MORE[6]. The second application is live video streaming. For this an actual application was developed which is able to stream videos over multiple tree structures using one of the congestion controllers.

3.1. Anonymous information sharing

In this section anonymous information sharing is motivated, one such anonymous overlay is presented (MORE[6]) and it is shown how to incorporate the congestion controller into this overlay network. This is not a trivial task as it would seem at first glance.

3.1.1. Introduction

To share information freely via email, (video) chats or files without being identified on the Internet (e.g. as source for Wikileaks(http://wikileaks.org)) one has to be anonymous. That means the IP address needs to be hidden. We can disguise our IP address by IP spoofing (Changing the sender address in the IP header). But if we want the node we share information with to be able to answer, we have to give this node some address to reply to. The solution to this problem is using relay nodes. We find some peers who are willing to forward our packets and return answering packets. This way the receiver only has the IP address of one of the relaying nodes and has to have control over the whole path to discover the IP address of the sender.
If it is sensitive information, we would not want the relaying nodes to be able to see the data we sent to the receiving node and to know who is communicating with whom. A popular approach then is to use Onion encryption[5]. There the sender adds an extra layer of encryption for each relaying node. Each relaying node can only remove one layer of encryption. Decrypting the packet reveals the address of the following relay node. The remainder of the packet remains unreadable because it is still encrypted by other layers. One relay node thereby only knows the previous and next node. The data changes (because it is decrypted by each node) from hop to hop and cannot be traced.
Applications like TOR[4] create one such path from sender to receiver over a number of relay nodes. Because they only build one path they are susceptible to traffic pattern

attacks like the one described in [29]. Motivated by this, the idea is to use different relay nodes for each packet. This is done in MORE[6]. We then have the problem this thesis tries to solve: We have numerous relay nodes available and want to build a path with them while avoiding congesting nodes. We have to somehow lean, either by the AIMD or the Q-Learning approach, the respective rates the nodes are able to deal with.
In MORE this is not done yet and random paths are built like in section 6.3.1. As we will see in the simulations (See Chapter 7) this gives very poor results and should be supplemented by a more sophisticated congestion control as described in Chapter 4 or 5.

3.1.2. Motivation for MORE

One reason for using a different path for each packet was already mentioned: Susceptibility of the single path approach to traffic pattern attacks. An attacker in a path can send flows over other nodes and even congest them. If then traffic in the path is decreasing there is a high probability that the node that was manipulated is a member of the path (See Figure 3.1). Using this technique an attacker can discover whole paths without being in control of a huge percentage of the network (see [29] for details).
MORE is immune to this kind of attacks, because of its dynamic nature. Each path is

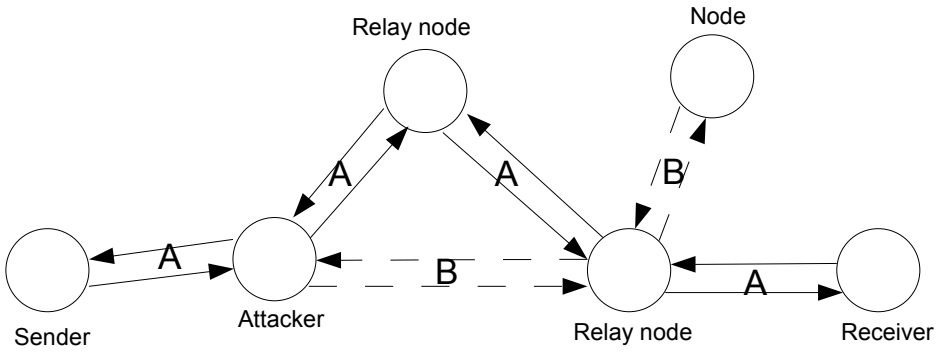


Figure 3.1.: The sender anonymously sends over three relay nodes to the receiving node. The attacker can probe a node by congesting it(B). If the flow from sender to receiver (A) diminishes the attacker knows that the node is part of the path.

different, so attacks which cannot instantly discern information but need some measurement time are not applicable. The traffic pattern attack described above is one such an attack.
It has other desirable properties as well: The random routing mechanism automatically balances load and reduces the impact misbehaving nodes have on throughput performance.

3.1.3. Technique

We assume a central element (central server, DHT, ...) with which information can be shared to be present, as well as some kind of public/private/shared key cryptography.

Then exchanging information anonymously can be done like this:

- **Publish receiver paths**
  Nodes who want to be able to be reached need to publish information on how they can be contacted. This is done by publishing receiver paths in MORE. The receiver first builds up paths the same way it is done in onion routing. While building up that path the receiver exchanges secrets with each participating node. Because this is done using the onion principle only the first node knows the IP address of the receiver and the shared secrets are only known by the receiver and the node the secret is shared with. The receiver then publishes these paths with his public key.

- **Sender packet construction**
  A node who wants to send something to someone anonymously gets a random receiver path using the public key of the node as identifier for the peer. The data it wants to send is encrypted with this public key, so only the receiver can read it. The sender then prepends the receiver path and then prepends headers for each node in a sender path in reverse order. After adding each single sender relay header the whole packet constructed so far is encrypted using the public key of the node the header was added for. This way only the nodes we designated as sender path members can access the respective headers. The structure of a packet as it leaves the sender is demonstrated in Figure 3.2.

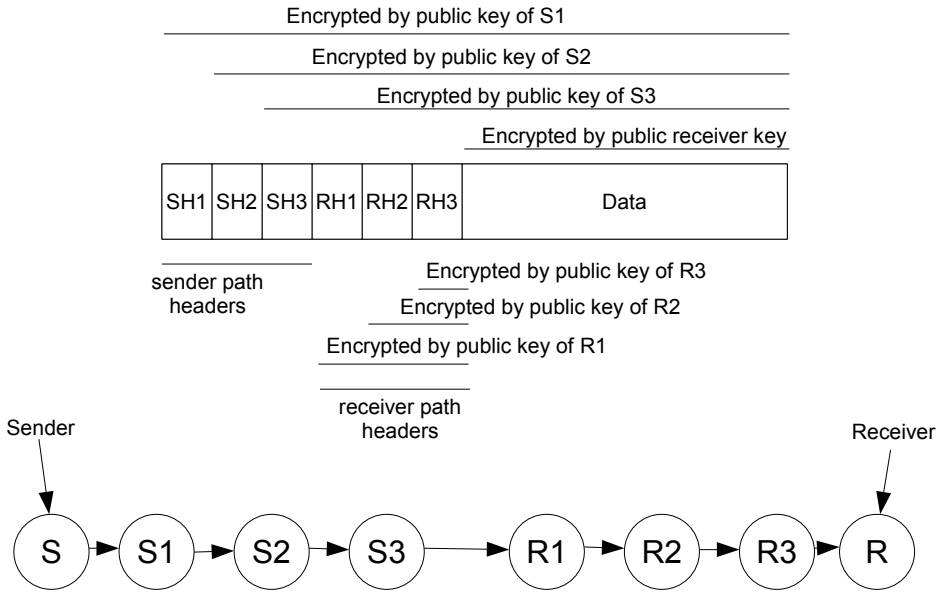


Figure 3.2.: Schema of a packet sent by MORE from sender to receiver.

- **Relaying in sender path nodes**
  The sender path relay decrypts the packet using its private key and can then access the header. There the address for the next node is stored. It removes the header and forwards the packet to the next node. Figure 3.3 shows the packet after it went through all sender path nodes.

- **Relaying in receiver path nodes**
  A receiver path node receives the packet and decrypts the receiver path header using its private key. There information about the next node is stored. It removes the header and encrypts the data section using the secret it shared with the receiver and sends it to the next node. Figure 3.4 shows the packet after it went through all

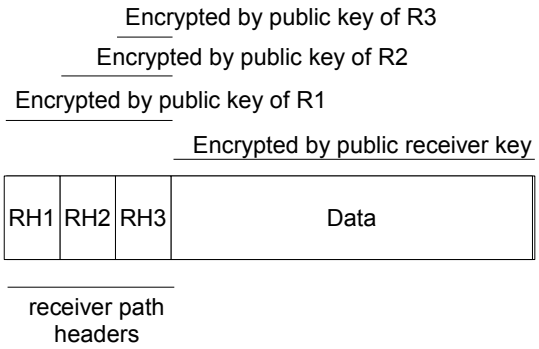


Figure 3.3.: Schema of the packet after it was relayed by all sender path nodes.

receiver relay nodes.

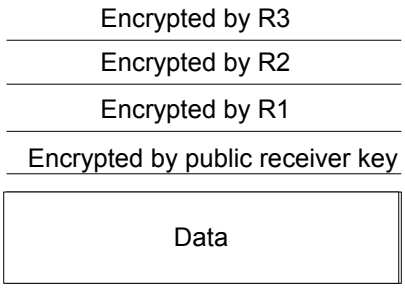


Figure 3.4.: Schema of the packet after it was relayed by all receiver path nodes.

- **Receiving the packet**
  The receiver receives the packet. Because he knows all the secrets shared with the receiver path nodes he can remove the layers of encryption added by the receiver path relays and then finally decrypt the data using its private key.

- **Replying**
  If the sender adds a receiver path consisting of the sender relays he used to the packet data, the receiver can use the same nodes the incoming packet used for constructing the path of the reply. The receiver uses the receiver path the sender added to the packet as receiver path of the reply and uses the receiver path nodes he got the packet from as sender path for the reply. If he adds another random receiver path to the packet as well the sender does not have to look up new receiver paths again if he wants to continue communicating.

This is just a short recap of how it is done. For details please look at [6].

3.1.4. Congestion Control

Congestion control can be done by requiring an acknowledgment to be sent for each packet that is received. If a sender sends a packet and does not receive an acknowledgment from the receiver, congestion occurred. This congestion can either occur in the IP layer or in the relays, if the bandwidth allocated for relaying packets is exceeded.
As sender we can only choose the sender path. Of the different receiver paths we only know the first relay. We can build up sender paths with nodes that we think are uncongested.

Likewise the receiver only publishes paths that he thinks contain only uncongested nodes. Depending on if the sender received an acknowledgment or not it adjusts the load of all nodes involved in sending that packet (that he knows of) accordingly, either by using the AIMD or Q-Learning approach (discussed in Chapter 4 and 5 and compared in Chapter 7). If for example it did not receive an acknowledgment, the sender normally reduces the load of the nodes involved and thereby uses them less often. The receiver does not receive a congestion signal for the receiver paths – the signal it gets if it acts as sender has to be enough – the congestion only works well if all nodes send and receive data. Nodes that only receive data are bad for the congestion control in this application.

Round trip time estimation

An important part of congestion control is round trip time estimation. We want an as exact as possible estimation of when an acknowledgment of a sent packet will arrive in order to timely detect congestion. This is difficult here, because we only have information about the nodes involved in the sender path available. As this is too little information about the whole path, we use the time we can measure between sending a packet to receiving an acknowledgment for that packet, only as means to estimate a general idea of how much time a packet needs for traversing the receiver paths of a receiver. But how do we know the time the packet needs for traversing the sender path?
Because the receiver has to answer with an acknowledgment to each packet he receives it is convenient that we add a receiver path to the packet we sent, so that, as mentioned above, the receiver can use this path to reply. If we use nodes in the sending path as receiver path nodes as well, we have to share secrets with them. If we share a secret with the last node in the sender path, we can get a pretty decent estimate on how long a round trip time from sender through the sender path and back is. If we then add the estimated receiver path time to that round trip time, we have an estimation of the round trip time from sender to receiver.

3.1.5. Conclusion

We have shown that the congestion control approaches developed in this thesis can be used to select uncongested peers for paths in MORE. This should improve throughput performance, as currently there is no such congestion control in MORE. We will see in Chapter 7 that this really causes very low throughput rates and should be avoided. Given the descriptions of how a congestion control algorithm can be incorporated into MORE it should be no problem to actually do it and thereby improve the performance of MORE greatly.

3.2. Video streaming

In this section a live video streaming application is motivated and described which uses one of the developed congestion controllers (Q-Learning congestion controller) to stream the data through various trees.

3.2.1. Introduction

A pure peer-to-peer application does not have a central control mechanism, but as we want to stream a live video stream it is obvious that some node has this stream first and is interested in distributing it. Peer-to-peer networks are often used to increase robustness. This cannot be a goal here, because we have one single point of failure – the node the stream originates from. The only thing we are interested in is to increase the capacity available for the stream by using freely available bandwidth in the clients. The only node with certain motivation is the source – it wants to distribute the stream. All other nodes can be potentially malicious.

Tree based streaming

In tree based streaming we build a tree to stream and distribute the content. It is often the case that a client cannot send the whole stream to another client because of limited outgoing bandwidth. Either the client is an interior node and has to forward the stream to at least one node or more or it is a leaf node and does not have to forward anything at all (See Figure 3.5). Another problem is that the tree's depth is increasing if there are only a few clients available who can handle more than one child. To rectify that problem the stream is split into different slices. Each slice gets its own tree. That way if a client has only bandwidth available to forward one slice to several children, it is interior node in one tree and in every other tree it is a leaf node. The difference in outgoing bandwidth between having one child more or less is smaller – we can scale the actually used bandwidth better and can get closer to the maximum bandwidth. Because we can use the bandwidth better the tree is less deep and the delay of the live stream is smaller.

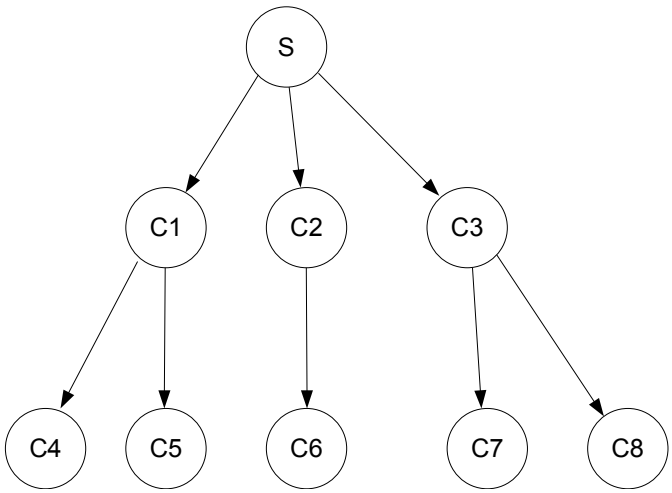


Figure 3.5.: Interiour node C3 has to forward to two other clients while the other leaf nodes C4-C8 have to forward nothing.

3.2.2. Motivation

There are currently two state of the art approaches to live peer-to-peer video streaming. Streaming with multiple trees (e.g. SplitStream[28]) or mesh based streaming (e.g. PRIME[26]) with mesh based streaming currently being favored, because it uses capacities better and is less susceptible to churn (see [27] for comparison). Both approaches, however, forget that in real life scenarios we have someone, that is interested in publishing that live video stream and willing to invest some resources in order to get this stream to the clients with acceptable quality. All other participants want to just watch the video stream, other interests of these participants are unclear. If they are willing to share the stream they received it is good for us, but we should prepare for clients not doing that, or find a way to enforce participation. Either way, giving important jobs to the streaming clients does not increase robustness, but decreases it. We never know, if they really will do what they should. So the streaming server should do the management of the nodes and repeatedly check if the clients really do what they should.

3.2.3. Technique

The proposed approach is now to use some part of the bandwidth available to the streaming server to explore how the nodes can help. Ideally that 'exploring bandwidth' should add redundancy to the actual streaming and be as efficient as possible. The other part of the available bandwidth is used to do the actual streaming using a tree based approach such as in SplitStream[28]. It can be seen as exploiting the information we gained with the other part of the bandwidth. [27] gives two reasons why the tree based approach is inferior to mesh based approaches: *(i)* content is mapped to a certain tree and *(ii)* nodes are interior nodes in only one tree and leaf nodes in all others. In this approach we know through the exploration estimations of bandwidths the clients are able to handle and estimations of the latencies between different clients. With this knowledge we can build trees where a client is as often an interior node as it is able to handle it and content is mapped to trees that can handle that content. So it should be performing better or as good as the mesh based approach given the streaming server gains enough knowledge about its clients.

Estimating the client bandwidths and round trip times

As already stated, we take some part of our bandwidth and use it for exploration. We do that by sending random clients content over multiple random clients. The motivation to send it over multiple hops is to gain as much information as possible with the bandwidth we can use. Every client of the streaming server builds up a TCP connection to announce the ports he wants to use and to receive how he has to participate. The streaming server sends pings regularly to the clients to see if they still exist. That way we can react fast to clients going offline. We can also measure the round trip time from streaming server to client with these pings. Because we have this information the last hop in the chain can send back acknowledgments to exploration packets directly back to the server like in Figure 3.6. We use the information if an acknowledgment arrived or not to update the estimates of the bandwidths the clients are able to handle using the congestion control algorithm developed in Chapter 5. We have to be careful to update the trees before we update the client bandwidths, because we want to know if the client is able to handle

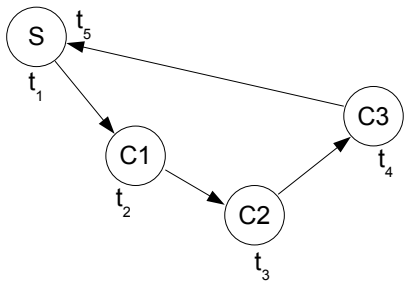


Figure 3.6.: The streaming server S sends a packet at time t_1 on a random path to estimating throughput and round trip times. Because the server already knows an estimate for $t_5 - t_4 = \mathrm{RTT}_{C3}/2$ through the direct connection to C3 $t_3 - t_2$ can be estimated as $((t_5 - t_1) - (t_5 - t_4))/3$

the bandwidth he has to forward by being an interior node in some number of trees plus the exploration packets the streaming server sends. If it is able to handle both, we can increase the bandwidth. If not, the bandwidth is decreased and the client is relocated to be a leaf node in a certain number of trees. This can be further refined by the clients sending negative acknowledgments to the streaming server if they did not receive content they wanted in time. That way the streaming server knows something failed in the parent of that node and can adjust the bandwidth. We still have to use exploration packets, because we have to estimate round trip times as well. We need them for the timeout values for the acknowledgments and for building efficient trees where clients with low latency are on top.

3.2.4. Outlook

An in-detail description on how this tree-based streaming was implemented will follow in Section 6.4 in Chapter 6. In Chapter 7 some trees produced by the streaming applications in an example setup will be shown.

Chapter 4.
4 Adaptive Increase, Multiplicative Decrease approach

In this chapter one of the two congestion control algorithms is constructed. It is based on AIMD (Adaptive Increase, Multiplicative Decrease). Though it seems simple, several problems with regards to the application with multiple parallel paths appear. These problems and their solutions are described in this chapter. Afterward it is shown how this congestion control algorithm was implemented.

4.1. Introduction

AIMD is the most widely used congestion control approach as it is used in TCP, which is wide spread and present in every modern operating system. This, of course, means that TCP is well tested under realistic conditions. AIMD has also been studied as a distributed solution to an optimization problem. Using analytic flow models it has been proven (e.g. in [30]) that AIMD(TCP Reno) is both fair and efficient in the equilibrium and that the equilibrium is reached, but they also discovered that this equilibrium is unstable if the link has high capacity or the round trip time is high. Papers like [31] propose (mathematically justified) solutions to this problem, but special care has to be taken, because the new algorithm has to be both fair to existing TCP implementations and implementable with the same kind of available information. Approaches like [16] seem to get the upper hand, as it is already implemented in Microsoft Windows, where increases in round trip delay are used as an additional indicator for impending congestion and in combination with traditional loss based congestion detection are used to adjust the sending rate. Since Linux kernel 2.6.18 the default TCP congestion control algorithm is Cubic TCP[32] where window growth follows a cubic function and can occur without congestion signal.
Using AIMD in our problem is straightforward. Instead of maintaining one rate that we increase/decrease adaptively/multiplicatively per link, we maintain a rate that gets adjusted by AIMD for each peer. Regardless of the straightforward adaptation to our problem, we cannot apply the control theory cited above, as, for example, we do not have one round trip variable per congestion controller but multiple ones, since each packet can come from an arbitrary chosen peer. Similarly, we cannot use round trip times as an indicator for congestion as we can only estimate upper bounds on delays between peers (more on that later on).

4.1.1. Simplifications

TCP was designed with very slow and limited devices in mind and as such uses some tricks to avoid e.g. floating point arithmetic. Additionally, because each packet has overhead, it uses other tricks to merge acknowledgments and packets. It further uses states like slow start, congestion avoidance and fast recovery, to converge fast to the optimal rate and recover fast after a packet loss. Some of these tricks can be avoided, because we are not interested in the last bit of performance. We just want to get a general idea of how good this approach is and if it is stable. We can then avoid using some of these tricks – others like the slow start mechanism are necessary to keep the approach competitive.

1. **Self clocking**
   TCP is most of the time implemented by self clocking. That means a variable *c_wnd* is maintained, that describes the current size of the congestion window. If *c_wnd* bytes are sent and on-the-fly no other packets can be sent. If an ACK is received, the size of the packet is subtracted from *c_wnd* and packets of that size can again be sent. This is called self clocking, because the packets are sent on acknowledgment arrivals and no additional timer is needed other than for acknowledgment timeouts. It is also ensured that there are only as many packets on the fly as packets fit in the current congestion window.
   A rate based approach, where instead of *c_wnd* the current sending rate is maintained is easier to understand, but more difficult to implement. Current operating systems run their scheduler around every 10ms. So if the rate is such that we have to sleep less then 10ms between sending two packets we have the problem, that we sleep too long if the operating system switches threads in the scheduler. This is a non-issue in the simulation, because it does not run in real time and uses timers with a high enough resolution. Because of this and its simplicity the rate based approach was selected for the simulation.

2. **Avoiding floating point arithmetic**
   For some calculations, like round trip time estimation, special forms were deduced which can be calculated by integer arithmetic. We can live with some performance loss and take the original formula using floating point arithmetic.

3. **ACK/Packet merging and piggybacking**
   To save some bandwidth outgoing acknowledgments can be piggybacked to outgoing packets or multiple packets or acknowledgments are merged into one to decrease the packet header overhead. This just saves a small amount of bandwidth and does not affect stability or fairness at all and as such would only distract from the central questions this thesis wants to answer.

4. **Slow start, congestion avoidance and fast recovery**
   TCP starts with a slow start phase where in each RTT time slice the size of the congestion window gets doubled by increasing it by one on each acknowledgment. Such a mechanism is needed for fast convergence to the equilibrium and as such is used here as well.
   After a packet loss occurred in the slow start phase TCP changes into the congestion avoidance phase with the last congestion window which could be transferred without packet loss. In this phase each time a whole window got transferred the size of the window is increased by the size of one packet. This is the equilibrium state and we

obliviously need this one.
Packet losses can be either detected by duplicate acknowledgments – that means the acknowledgments for following messages arrive without having received an acknowledgment for the current message – or by an acknowledgment timeout where it took longer for an acknowledgment to arrive then the estimated round trip time plus some margin. As the acknowledgment timeout only occurs in extreme congestion TCP Reno starts again with one maximum segment size as the congestion window into the slow start phase. On duplicate acknowledgments the congestion window gets halved (fast recovery). It is assumed that the switch to slow start only has to be used in severe cases of congestion, we can leave out that mechanism and only implement fast recovery. As only fake data is sent in the simulation, we do not have to resend data and instead can simply send new data. Because of this the only disadvantage of not using duplicate acknowledgments as congestion signal is delayed reaction time which should not influence the overall stability – we can use acknowledgment timeouts as our only indication of congestion. If such a timeout occurs we can appropriately decrease the rate multiplicatively.

4.1.2. Motivation

The exponential backoff in AIMD is directly motivated by a link model (see [18]). We consider a network where there is no congestion and L_i is the average link load at time i and N denotes the average arrival rate of packets. By [18] it holds:

$$L_i = N \tag{4.1}$$

As the links are not congested they can forward the traffic in time step i and the load is constant.
If we consider a congested network:

$$L_i = N + \gamma L_{i-1} \tag{4.2}$$

At each point in time, citing [18], we have our original load plus a factor which is leftover traffic from the last time step plus effect of this leftover traffic like retransmits. γ influences the strength of the congestion. [18] says this gives an exponential increase of queue lengths and leads to an eventual breakdown. [33] is cited as a source of the assumption that queue length increases exponentially and as such the congestion controller has to back off at least exponentially as well. The problem is that the cited paper assumes the congestion to be caused by a shared medium e.g. a shared Ethernet connection or ALOHA without central control. Indeed it has been proven (in [34]) that nothing less than exponential backoff results in stability in such a broadcast medium.
In todaySs Internet such shared unmanaged links are rarely seen. Even the occasional hub on the end user side has vanished. The exponential backoff is not necessary anymore and only degrades the throughput performance. As mentioned in the introduction, if a router has certain capacity it can always forward this amount of traffic especially if its buffer is full. This does not mean that round trip times will not increase or certain flows will not "'break down"' – effects which can be mitigated by a right queuing strategy [1].

[1] Popular queuing strategies: Drop Tail where after the queue buffer is full new arrivals are simply dropped and Random Early Detection (RED) where packets are randomly dropped with increasing probability if the buffer gets full (The second strategy will be more fair). See [35].

[36] tries to show a migration path to an exponential backoff-less TCP, but frankly – who wants to be the culprit building an operating system that is unfair to current TCP flows – and it is obvious that exponential backoff-less flows will always be unfair to flows that do an exponential backoff. Regardless, we now have two motivations for using AIMD as a congestion control algorithm:

1. It is well tested and in use for about thirty years. It is stable, easy to implement and with good performance.

2. If we want to be fair to existing TCP flows we have to do an exponential backoff on congestion – which we do if we select the AIMD approach.

If we only care about the throughput of our overlay network we can remove the exponential backoff and think about other strategies. This is why the approach with Q-Learning was done.

4.2. AIMD Congestion Controller

In this section the details that in their sum form the AIMD Congestion controller are shown. Several key problems are presented together with their solution, such as round trip time estimation or the slow start mechanism.

4.2.1. Round Trip Time estimation

One important part of any congestion controller is how it gets its congestion signal, which indicates congestion occurring. One signal that is always present and is, therefore, often used is packet loss. If we lose a packet we can assume that it was lost by congestion. We would want to receive this congestion signal quickly after the congestion occurred. For that purpose we need an as exact as possible estimation for the time at which an acknowledgment for a sent packet gets returned. We have data available from which we can learn this round trip time: Each time we send a packet we get an acknowledgment. The packet was routed through a number of nodes. We know which nodes. If we save the time we sent the packet and the nodes it was sent through we can estimate the time spent between the different hops (See Figure 4.1). If we are estimating the most probable next round trip time between two nodes we want two things: First more recent RTTs have to get more weight than older RTTs and second we want to really be sure that an acknowledgment is able to be returned in that interval.
We can reach these goals by calculating a so called SRTT, also used in TCP. If L is the estimated round trip time between two nodes and l is a more recent measurement for L we can estimate a new mean by:

$$L = (1 - \gamma)L + \gamma l = L + \gamma(l - L) \tag{4.3}$$

As we can later see, it is very similar to the update rule used in the Q-Learning Chapter, and indeed it was directly motivated by stochastic descend[3]. γ steers how much weight is given to more recent samples and is usually selected around 0.1.
To reach the goal that the estimation should never be too small, we have to calculate the

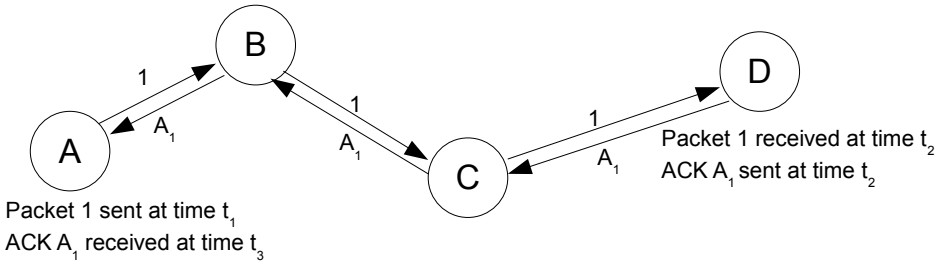


Figure 4.1.: A Packet gets sent at time t_1 with two hops (B,C) to D where it arrives at t_2. D sends an ACK over the same nodes, which A receives at t_3. A can measure $R = t_3 - t_1$. To estimate the latencies (times two) between the nodes we can take $R/(h+1)$, with h the number of hops between A and D. This is far from perfect as this way we can never exactly estimate the latency with the lowest or highest value regardless how we route the packet. To optimize this could be an area of further study.

variance σ^2 of L. We can estimate an upper bound V for σ^2 computationally efficiently by

$$V = V + \gamma(|l - L| - V) \tag{4.4}$$

The time T in which the acknowledgment usually returns is then given by

$$T = L + \alpha V \tag{4.5}$$

With α usually chosen around 4.
We want to estimate the time a packet needs from one node over several nodes to another and back again. If A is our source node, B the target node and H^i are the hops in between with h the number of hops and with L_{kj} denoting the SRTTs between the nodes and V_{jk} the estimated variances, the time within which the ACK for a packet should be received can be calculated as:

$$T = L_{AH_0} + \alpha V_{AH_0} + \sum_{i=1}^{h-1}(L_{H_iH_{i+1}} + \alpha V_{H_iH_{i+1}}) + L_{H_{h-1}B} + \alpha V_{H_{h-1}B} \tag{4.6}$$

4.2.2. Rate based approach

Instead of a congestion window based approach a rate based approach is taken. Packets are sent with a rate of r typically measured in bytes per second. Usually using that approach one has to handle difficult timing issues, because if many packets are sent at once buffers can overflow that would not overflow if they were sent uniformly distributed over a certain time slice. This is not an issue at our simulation, as we can exactly time sending single packets and are not too concerned about performance loss.

4.2.3. Rate update rules

TCP halves the congestion window on a packet loss and increases it by one if a whole congestion window was successfully sent without loss. We want to mirror this behavior in our AIMD approach to reach fairness with competing TCP streams. So if a packet was sent like in Figure 4.1 from A over some hops H_i with rates R_i to D and we successfully

receive an ACK for this packet, we update the rate for the hops like this:

$$R_i = R_i + \frac{\alpha M}{L R_i} \tag{4.7}$$

with M being the (network dependent – typically 1500 bytes) maximum segment size and L being the Round trip time from A to H_i as in equation 4.3. α should be selected as 1 if we want to be TCP like.
As for each packet

$$\frac{\alpha M}{L R_i} \tag{4.8}$$

gets added. And we have

$$\frac{R_i L}{M} \tag{4.9}$$

packets per round trip time. In one RTT

$$\frac{\alpha M}{L R_i} \frac{R_i L}{M} = \alpha \tag{4.10}$$

gets added. Which is the same as in TCP where each RTT one packet gets added in the congestion avoidance phase.

The update rule for a packet loss/timeout is:

$$R_i = \beta R_i \tag{4.11}$$

With $\beta = \frac{1}{2}$ the equivalent to TCP where the congestion window gets halved if a packet gets lost.
The rate is changed immediately after the event occurred following [3] although [19] claims (somewhat unmotivated) that this causes oscillatory behavior.

4.2.4. Slow start

To speed up the convergence to the equilibrium rates the slow start mechanism of TCP is added. This does not affect the behavior at equilibrium points.
In the slow start phase for each incoming acknowledgment one additional packet gets sent each round trip time. This means the update rule changes to:

$$R_i = R_i + M \tag{4.12}$$

The slow start phase is left as soon as a single packet is detected as lost with a rate of $R_i = R_i/2$. In the experiments handling this rate based was a bit tricky. If multiple packets got lost the rate is halved multiple times, which is not what we want as we know it worked with $R_i/2$. The solution was to save the rate at which the packet was sent and if the congestion controller was in the slow start phase while the packet was sent. The saved rate is then taken as input for the update rule and the rate is only updated for the first packet that times out.

Chapter 5.
Reinforcement Learning Approach

In this chapter, the second congestion control approach is presented and constructed. It is based upon Q-Learning, which is often used as a simple, yet powerful, way to do reinforcement learning – to train a controller to do actions for which it receives the most reward. The state with the most reward in a congestion controller is when it has selected the maximal rate with no packet loss. A general description of reinforcement learning, specifically Q-Learning is given. Then the Q-Learning congestion controller is developed. Some problems that occur are described and their solutions are presented. It is shown how it was implemented into the simulation.

5.1. Motivation

We have already shown in the previous chapter that AIMD with its exponential backoff is outdated, because it is motivated by the assumption that the congestion occurs in a shared medium. As in our problem congestion mostly does not occur in the links but in the nodes and we have total control of the queuing strategy used, we can definitely say that it does not occur in a shared medium. Let us look at what happens at congestion at a node. If the node is uncongested it can forward all its incoming packets. If L_i denotes the load of one node at time i and N the number of incoming packets at time i and n the number of packets that can be forwarded in one time step. Then

$$L_{i+1} = N + \max(0, L_i - n) \tag{5.1}$$

as in every time step. n packets can be processed and N new packets arrive. If $N \leq n$ and $L_0 = N$:

$$L_k = N \tag{5.2}$$

If $N > n$ the network is congested and

$$L_k = N + k(N - n) \tag{5.3}$$

That means the queue length is increasing by $N - n$ each time step. After some time the queue will be full. Some packets will get dropped and the congestion controllers will have to react, because dropped packets have to be resend and thereby reduce the throughput. But all the congestion controllers of the flows through that node have to do is to reach $N \leq n$. If we have a congestion controller which can only increase its rate by one packet each round trip time and the buffer at the congested node is so small that it fills within

this time frame, all the congestion controller has to do is decrease its rate by one packet. We can demonstrate this by an example. If there is only one flow going through that node either $N > n$ or $N \leq n$. If $N > n$ only $N = n + 1$ could be the case, if the congestion controller immediately reacts – and decreasing the rate would remove the congestion. If j flows are going through that node only $N \leq n + j$ could be the case, assuming that the congestion controllers react immediately, and each flow throttling back by one packet would remove the congestion.
Motivated by this, a congestion controller is built that can only increase and decrease its rate by one packet each round trip time and learns which action causes congestion and which only improves throughput.

5.2. Reinforcement Learning in General

Reinforcement learning is an area of machine learning. The problem reinforcement learning has to solve is more difficult than the classical task of classification. Usually, some knowledge of the task already exists in the form of training data– data of which the result (class) is known. In reinforcement learning, an agent receives a signal – called a reward – if something was good or bad. The agent may not even get this reward instantly. This is somewhat inspired by how e.g. a mouse acts if it has to find a lump of cheese in a labyrinth. The mouse only gets the reward if it reaches the cheese, not before that. If we put the mouse into the same labyrinth repeatedly and do not change the position of the cheese, a smart mouse will have memorized the layout of the labyrinth and, if it has gone through the labyrinth often, it may even know the shortest route to the cheese. The mouse – the agent – has to learn rules with which it can maximize the reward it gets. Because unknown combinations of actions can yield unexpected rewards – an unmapped part of the labyrinth may yield a shorter path – the agent has to try different combinations by trial and error.
What we need for an agent for reinforcement learning:

1. Some kind of reward feedback. This means we have some kind of signal which says an action was good or bad (this would be the mouse finding the cheese).

2. States and actions. The agent is in a certain state and can choose to do certain actions. Then it gets the reward. Choosing an action changes the state (the state would be the position of the mouse in the labyrinth; the actions would be where to go next).

As the agent knows nothing to start with, it can only learn from experience. This is what reinforcement learning does. The problem that instantly arises is balancing exploration and exploitation. On the one hand exploiting previously gained knowledge gives a sure amount of reward, but on the other hand exploring new states through previously not executed actions may yield even higher reward – or none at all. This is of course difficult to balance, even more so if the reward given in certain states may change. Reinforcement learning means learning while interacting with an environment. We can now identify some elements of a reinforcement learning system:

1. Policy
   The Policy defines how the agent behaves given what state it is in.

2. Reward function
   The reward function looks at the state the agent is in and depending on that state gives reward. Practically the reward takes the form of a scalar. The scalar is big on "'good"' states and small or negative on bad states.

3. Value function
   The Value function is not the immediate short term reward but rather the reward the agent gets in the long term. It is the reward the agent can collect starting from the current state onwards. One can see it as a prediction of future rewards. If we had this function we could make perfect decisions and would not have to explore at all. As it is we would have to go through all states from all states. Those are in most applications infinitely many – we can only estimate this value function.

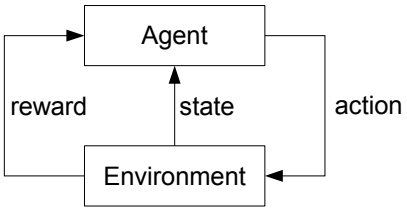


Figure 5.1.: The agent interacts with the environment by doing actions. It gets back a reward and a new state.

As one can see in Figure 5.1 the agent interacts with an environment. It does some actions and thereby changes it state, then it gets reward which depends on that new state and information about the new state itself. We hereby assume that the state has Markovian properties and that the process we want to control is a Markovian process. This means that we can build a finite number of states with each one describing exactly all relevant things about the situation the agent is in. The information about which state we are in fully satisfies our need for knowledge, that means additional information about how we got to this state is redundant – all we need for decision making is hidden in that state variable. The rewards in future states do not depend on which combination of actions brought the agent there.

5.3. Q-Learning

The technique for approximating the Value Function chosen here is Q-Learning. It uses the values saved in a Q-Table as approximations for the Value Function. Given a state s_t at time t a chosen action a_i, next state s_{t+1}, the reward r_{t+1} and π denoting our current policy[1], the Q update rule would be as follows:

$$Q^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q^{\pi}(s_{t+1}, a) - Q^{\pi}(s_t, a_t)) \tag{5.4}$$

[1]The policy says which action to take in which state

As one can see this formula follows a basic scheme every such technique exhibits:

$$newEstimate \leftarrow oldEstimate + stepSize * (target - oldEstimate) \tag{5.5}$$

Our agent chooses $\max_a Q^\pi(s,a)$ as action a to take in state s. Then the target is the return in state s_{t+1} plus the estimated return the agent will get in the future by following the current policy.

5.3.1. Detailed motivation for Q-Learning

If we had an infinite number of agents performing an infinite number of actions, we could exactly calculate our value function and thereby our perfect policy $*$. This would be given by

$$Q^*(s_t,a_t) = \max_{\{a\}} \frac{1}{N} \sum_{k=0}^{N} r_{t+k+1} \quad N \to \infty \tag{5.6}$$

$$= r_{t+1} + \max_a Q^*(s_{t+1},a) \tag{5.7}$$

Which is the best average of all rewards in the future. The rewards are caused by the sequence of actions $\{a\}$ the agent does. This is why the maximization is done over the actions not the rewards.
We have two problems here:

1. It is assumed the problem we try to solve with reinforcement learning does not change. But this is the case in our congestion control problem.

2. We cannot calculate an infinite sum. Even if we could calculate it (if we have infinite time) we could do a finite amount of bad actions given that we do infinitely many best actions and still get the "‘best"’ policy. This sum does not converge.

The idea now is to give short term rewards higher influence while still considering long term future rewards, which is called discounting:

$$Q^*(s_t,a_t) = \max_{\{a\}} \sum_{k=0}^{N} \gamma^k r_{t+k+1} \quad N \to \infty \tag{5.8}$$

with $0 < \gamma < 1$. If $|r| \leq a$ then

$$\sum_{k=0}^{N} \gamma^k r_k \leq \sum_{k=0}^{N} \gamma^k |r_k| \leq \sum_{k=0}^{N} \gamma^k a = a \sum_{k=0}^{N} \gamma^k = \frac{a}{1-\gamma} \quad N \to \infty \tag{5.9}$$

So as long as the rewards are bounded by some value a we have convergence and different strategies are really comparable.

As above this can be rewritten by pulling out the first element of the sum:

$$Q^*(s_t, a_t) = \max_{\{a\}} \sum_{k=0}^{N} \gamma^k r_{t+k+1} \quad N \to \infty \tag{5.10}$$

$$= \max_{\{a\}} \left(\gamma^0 r_{t+1} + \sum_{k=1}^{N} \gamma^k r_{t+k+1} \right) \tag{5.11}$$

$$= \max_{\{a\}} \left(r_{t+1} + \gamma \sum_{k=0}^{N} \gamma^k r_{t+k+2} \right) \tag{5.12}$$

$$= r_{t+1} + \max_a Q^*(s_{t+1}, a) \tag{5.13}$$

If we now only have an estimation Q^π for the perfect value function Q^*, we can use the above formula as the current target for our intuitive approach 5.5 and get the Q-Learning update rule 5.4. In the limit our estimated policy π should then converge to $*$.

5.3.2. Balancing exploration and exploitation

As already mentioned a big problem is finding the balance between exploitation of knowledge we already have about rewards and doing unexpected stuff and thereby perhaps gaining knowledge about even higher rewards. In practice this is often done using a simple ϵ-greedy strategy. This means we select with probability ϵ $0 \leq \epsilon \leq 1$ a random action and with probability $1 - \epsilon$ we follow our policy:

$$\begin{cases} \max_a Q^\pi(s, a) & \text{if } r < \epsilon \\ rnd(\{a_s\}) & \text{else} \end{cases} \tag{5.14}$$

With r being a random value with $0 \leq r \leq 1$ and $rnd(\{a_s\})$ giving back a random action that is possible in state s. The ϵ-greedy strategy selects each action with equal probability when it explores. This is sometimes bad, if some states generate low rewards. In those cases a softmax-action selection could be preferable. There, previous to each state transition, each action a gets selected with probability:

$$\frac{e^{Q^\pi(s_t,a)/\tau}}{\sum_{i=1}^{n} e^{Q^\pi(s_t,a_i)/\tau}} \tag{5.15}$$

if there are n actions available in state s_t. $\tau > 0$ regulates how hight the bias towards higher value estimates is. If τ is big higher differences in the value estimates do not cause that much difference in the different selection probabilities. If $\tau \to 0$ the agent does greedy action selection which means he always selects the action with the highest Q-Value and does no exploration at all.

5.4. Q-Learning Congestion Control

As the terminology of reinforcement learning was already explained in the previous section, we can now map from general reinforcement learning to our congestion control problem.

1. Agent
   The agent is our congestion controller

2. States
   The state of the agent is a discrete representation of the current congestion window or sending rate we use for congestion control. Because of the rate based approach the state represents the number of messages that can be sent per second. This means we send a maximum of s_t packets each measurement interval. This also means that the current state is just a positive integer.

3. Action
   The actions an agent can do in each state (but the upper and lower bound) are increase,stay and decrease. Increase and decrease modify the state and with that the sending rate by 1 respectively while stay does not modify the state at all (self transition). First the action stay was let out, but experiments confirm more stability if it is used.

4. Reward
   Positive reward is given if an acknowledgment for a packet arrives. A negative reward is given if an acknowledgment does not arrive within the mean round trip time plus four times the variance of the round trip time.

Problems:

1. Delayed reward information
   With that mapping we get delayed reward information. That means, we do not immediately get reward after we chose an action, but get it usually within the mean round trip time. It is important we only do state transitions if we have received the reward.

2. Incentives for higher transfer rates
   If we only give positive reward if there is no packet loss and negative reward if there is, the agent will have no incentive for going to higher states and respectively higher transfer rates. Higher states have to yield higher rewards then lower states.

3. Initial values
   We do not know which state to choose as initial state and how the initial Q-table should look like.

4. Unsaturated connection
   If the connection is not saturated we will probably only get positive reward even if we are in a state where we would get negative reward on a saturated connection. This is an inherent problem with all congestion control algorithms. We could either ignore that and take into account some lost packets or slow down or stall state transitions if the connection is not saturated.

5. Exploration strategy
   The exploration strategies have to be evaluated and one has to be selected.

5.4.1. Delayed reward information

We get our reward information when we receive acknowledgment packets or if there is an acknowledgment timeout, which is in the worst case after $RTT.mean + 4RTT.var$ seconds[2]. We would want to have this information before we do our state transitions. This is simply done via acknowledgment counting. For each acknowledgment we receive or that timeouts a counter is incremented. If the counter reaches the value of the current state (which is the packet sending rate), we reset it and do the state transition. If s_t is our current state we have received s_t acknowledgments or acknowledgment timeouts by the time we select our action and change our state and by that time we hopefully have enough reward information. $r_d = -1$ was selected as a 'reward' for a timeout. The reward for an ACK is further calculated in the results chapter as $r_u = 0.017$. The only condition we have to meet, so that the agent really avoids congestion is $-r_d \geq r_u$, otherwise the agent would have the same reward in states where there is no congestion as in states where there is congestion, due to effects explained later in the next section.

5.4.2. Incentives for higher transfer rates

The agent need incentives for higher transfer rates. This means rewards have to be higher for greater transfer rates than for lower ones. This is done by scaling the rewards by the current rate. This means the reward for an acknowledgment becomes $r_u s_t$ and the 'reward' for an ACK timeout becomes $r_d s_t$. Now we can see why we need the condition $-r_d \geq r_u$: If the agent experiences no congestion in state s_t he gets $r_u s_t$ reward (he gets $r_u s_t$ reward for s_t successfully sent packets, that gets divided again by s_t because its an arithmetic mean, that means $\frac{r_u s_t^2}{s_t}$). He increments the state. The current state is now $s_{t+1} = s_t + 1$. If then there is one packet lost in the next interval he gets

$$\frac{r_u(s_t+1)s_t + r_d(s_t+1)}{s_t} = \frac{r_u s_t^2 + r_u s_t + r_d s_t + r_d}{s_t} \tag{5.16}$$

reward. If $-r_d \leq r_u$ and $(r_u + r_d)s_t > -r_d$ then

$$\frac{r_u s_t^2 + (r_u + r_d)s_t + r_d}{s_t} > \frac{r_u s_t^2}{s_t} = r_u s_t \tag{5.17}$$

Which means given the rate is high enough (and thereby s_t) the agent would get higher reward even if there is congestion. To avoid congestion completely $r_u + r_d \leq 0$ should be met. Which gives $-r_d \geq r_u$. This means the agent should get less positive reward than negative reward.

5.4.3. Initial values

We have an initial state and initial values in our Q-Table. To get an idea how these should be set we can look at e.g. TCP, where on the beginning of the slow start mechanism only one packet is sent. For each successfully sent packet the number of packets that are sent is increased by one. This is an exponential growth, as during each RTT interval the number of

[2] the 4 is an adjustable value and is just chosen here as an example

outgoing packets gets doubled. On congestion TCP starts with half the congestion window into the congestion avoidance phase.
We can extract two things here:

1. As we do not know anything about the optimal sending rate, we have to start with the minimal value. This means $s_0 = 1$.

2. The rate should be increased very fast until congestion occurs. Then the rate should be lowered and further increase should not occur immediately.

The second goal is reached by implementing a fast start mechanism into the Q-Learning congestion controller. If the controller just started and there was not any ACK timeout yet, for each received ACK we set

$$\begin{cases} Q^\pi(s_t, up) = r_u s_t \\ Q^\pi(s_t, stay) = 0 \\ Q^\pi(s_t, down) = 0 \end{cases} \tag{5.18}$$

and increment the state by one ($s_{t+1} = s_t + 1$). This means we have exponential growth of the sending rate like in TCP at the slow start phase.
If we have an ACK timeout at state s_t all Q-Values of states $s = s_t, s_t + 1, s_t + 2, ...$ are set to

$$\begin{cases} Q^\pi(s, up) = 0 \\ Q^\pi(s, stay) = 0 \\ Q^\pi(s, down) = -r_d(s - 1) \end{cases} \tag{5.19}$$

All Q-Values of states $s = s_t/2, s_t/2 + 1, ...s_t - 1$ are set to

$$\begin{cases} Q^\pi(s, up) = 0 \\ Q^\pi(s, stay) = 0 \\ Q^\pi(s, down) = 0 \end{cases} \tag{5.20}$$

and the fast start phase is left with $s = s_t/2$ the current state.
This reflects the knowledge we have gained at this point:

1. The congestion occurs somewhere between $s_t/2$ and s_t. We do not know anything about the expected rewards in this area, so we set it to zero.

2. Congestion occurred in state s_t, so the expected rewards in states including and above s_t are biggest for the *down* action.

5.4.4. Unsaturated connection

We do not have the problem with unsaturated connections anymore as we only modify the current state after having received s_t acknowledgments or acknowledgment timeouts. This means if the connection is not fully saturated states are switched with lower frequencies. We could also easily do no state transitions at all if the connection is not saturated, or do only state transitions if the connection is saturated or a packet was lost (acknowledgment timeout).

5.4.5. Exploration strategy

This congestion control problem is non stationary. Another peer can suddenly start to use a node – then the perfect sending rate the agent previously had is not perfect anymore. The agent then has to decrease the sending rate accordingly. It is clear that at every moment in time it is equally likely that something like this happens. This means we should not use an exploration strategy where exploration probabilities diminish over time. We best use the simple ϵ-greedy exploration strategy.
After s_t acknowledgments or acknowledgment timeouts are received and if the agent is not in the fast start phase, the action is selected like this:
($r \in [0, 1]$ is randomly selected between 0 and 1)

$$\begin{cases} rnd(\{up, stay, down\}) & \text{if } r < \epsilon \\ \max\limits_{a \in \{up, stay, down\}} Q^{\pi}(s_t, a) & \text{else} \end{cases} \tag{5.21}$$

Chapter 6. Implementation

In order to better judge and compare the results two additional congestion controllers have been developed. One is a non-distributed approach and should, therefore, give next to optimal results. The other one does no congestion control and just uses random paths.

6.1. AIMD Congestion Controller

In this section it is shown how the AIMD congestion controller was implemented, which problems had to be solved and how it was embedded into the simulation framework that was used.

6.1.1. General information

The congestion control method was implemented using C++ and the Omnet and Oversim[37, 38] frameworks.

6.1.2. Rate handling

Since it is in a simulation, the rate handling is relatively easy. Rates are measured in bytes per second. We added timers that for each node reset the current rates $R_c$to zero every second. If we want to limit rates, we simply store the limit R_l and each time we want to send something look if the current rate (that gets reset every second) with the additional payload is smaller than that limit. This can only be done in a simulation, because it leads to bursty behavior, which would lead in reality to buffer overflows in routers on the path or the target node.

6.1.3. Finding a suitable path

Given a source A and a target node B and a message with size l, we want to find a suitable, uncongested path with h hops from A to B. We do that by following algorithm:

1. Collect all nodes K_i where $R_c + l \leq R_l$ and $K_i \neq A \wedge K_i \neq B$

2. For $i = 0, ..., (h-1)$: Select a random node H_i from $\{K\}$ and set $K = K \backslash \{H_i\}$

3. Return $(A, H_0, ..., H_{h-1}, B)$ as the path

6.1.4. Using the Oversim framework

The simulation is done using the Oversim framework. Oversim is somewhat specialized in simulating DHT overlays. To use the underlying structure of Oversim we have to fit the congestion controller to general DHT ideas.
One idea is different lookup strategies. Namely (full) recursive and iterative lookup, as shown in Figure 6.1. What we want for our simulation is something like the full-recursive

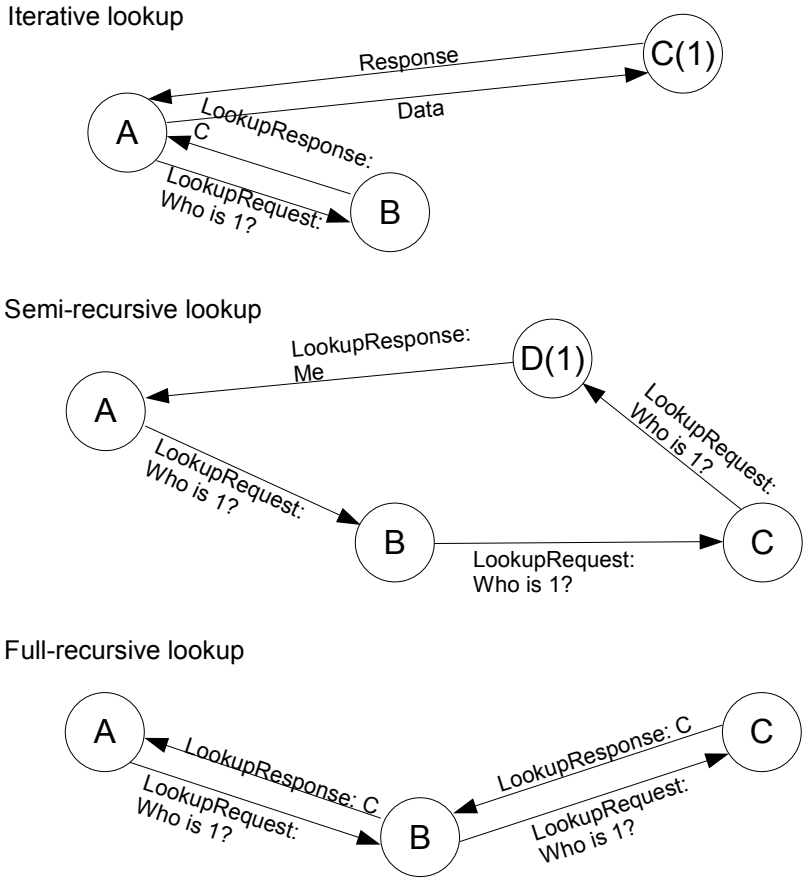


Figure 6.1.: Different lookup strategies in Oversim. Iterative, semi-recursive and full-recursive lookup.

lookup with data attached (see Figure 6.2). It was possible to implement this kind of flow using the full recursive lookup from Oversim.

Because of the DHT idea Oversim uses keys to abstractly identify nodes. We introduce a one to one mapping of ids to IP addresses, as we do not need this abstraction for the simulation.

If an application layer in Oversim wants to send a message recursively to another peer it simply sends this message to its own overlay layer. There the function *findNodes* decides

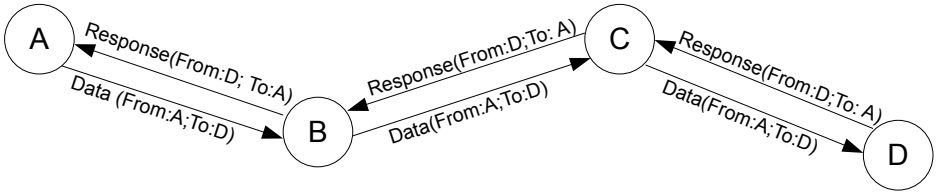


Figure 6.2.: General data flow we want. Node A sends data over nodes B and C and gets back some response through the same path.

where this packet goes to like this:

1. If the packet came from the application layer we set the current hop h_c of the packet to one (h being the desired number of hops) and use the algorithm from 'Finding a suitable path' to get a path $P = (A, H_0, ..., H_{h-1}, B) \Rightarrow P_0 = A, P_{h+1} = B$, which we store together with the current hop in the packet.

2. For every packet we get (including those coming from other overlay layers than our own) we first look if our current rate plus the message size is smaller or equal the maximal rate. If not we discard it.

3. We recover the path and the current hop from the packet and look at the address $G = P_{h_c}$.

4. If G is our own address we are finished and we send an ACK along the path $P_A = (B, H_{h-1}, ..., H_0, A)$ and forward the packet to the application layer.

5. If G is not our address we increment h_c by one and forward the packet to P_{h_c}.

6.1.5. Saturated connections

For some of the simulations we need saturated connections. For that we add an additional function into the overlay which tests if there are h (number of desired hops) nodes with enough rate left for a packet of size l and return false if not and true otherwise. If the application layer now asks this function if there are still unsaturated nodes and receives a negative feedback it can reschedule all other messages after the rates in the overlay will be reset.

6.2. Q-Learning Congestion Controller

As the AIMD-Congestion Controller the Q-Learning Congestion Controller was implemented and tested within the Omnet/OverSim framework. It is written in C++. It uses the same Latency estimation and timeout mechanisms as the AIMD-Congestion Controller (see 4.2.1). The general approach how the rates are handled, suitable paths are found and the Oversim framework is used, can be reused too (6.1.2-6.1.5). In the code this is done using inheritance (Both the AIMD and Q-Learning Congestion Conroller inherit from the same class which implements the basic stuff which both need).

6.2.1. Reward state information

In rare cases, because we have delayed reward information, we could already have switched the state, if the ACK arrives. The state in which the message was sent differs from the state in which the ACK (or ACK timeout) was received. The solution to this problem is saving the state and last action for each sent message. On an ACK or ACK timeout we can then recover this information with the message id (which we need for the ACKs anyway). The reward is not applied directly but saved for each state. If a state transition is about to occur, the arithmetic mean of all collected rewards is calculated and applied to the Q-table with the Q-Table update rule (5.4).
Motivation:
If we updated the Q-Values each time we got some reward (ACK or ACK timeout), the rewards themselves would be discounted, as the Q-Values are discounted like in (5.8). We would want to avoid that, as then early ACK timeouts would, ultimately, not give the same reward as late ACK timeouts. Intuitively, we would want each reward to have the same influence. This is the case if we collect them and apply an arithmetic mean just before the state transition.

6.2.2. Q-Table

The Q-Learning method used here works on discrete states and discrete action and as such is easy to implement. Because in real life we can use the rate of sent packets instead of the real sending rate, there are relatively few states. We can calculate the maximum number of states by dividing the outgoing link capacity (this information should be available through the operating system) by the maximum transition unit. The MTU is usually 1500 bytes for Ethernet. On a 100 MBit/s link we would then get a maximum number of 8739 states. As for each state there are 3 actions and each of them has a Q-Table entry and single precision should be enough, the congestion controller needs only about 100kB of memory for each peer. This can of course be further optimized by having a sparse Q-Table where states only touched by the fast start and not used at all are omitted.

6.3. Additional Congestion Controllers

To be able to further classify results two more congestion controllers were implemented: One that does nothing and simply selects random paths and one that should give close to perfect results, because it assumes there is a central manager, that has all information.

6.3.1. Random ‘congestion controller’

The random congestion controller should not have this name. It just selects for each packet that needs to be sent a random path to the target and does no further congestion control. It does not even need acknowledgments.
If it were a normal congestion controller we would assume the throughput to be maximal, with a very high drop rate due to congestion. But as we have a multi-path problem something like in figure 6.3 can happen. A Packet with a poorly chosen path can use up

bandwidth at the hops without reaching its target if it gets dropped when it has nearly reached its target. Thus the random 'congestion controller' gives us a worst case scenario with which we can compare the results of our real congestion controllers. This 'congestion controller' is sometimes called BasicCC in several plots later on.

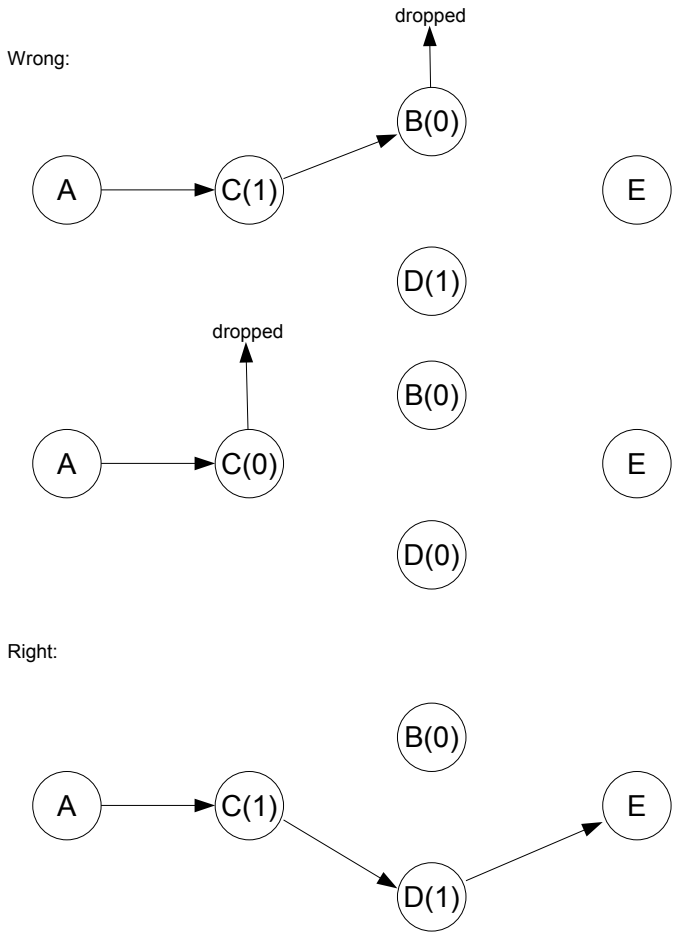


Figure 6.3.: A wants to send packets to E. In the brackets the number of allowed forwards this time slice in packets is denoted. If A selects the wrong peers as hops not only does the packet get dropped, but the bandwidth in C gets used up. Then no packet can be sent to E at all, because they all get dropped (if we want two hops). If A selects the right path, one packet can be sent to E.

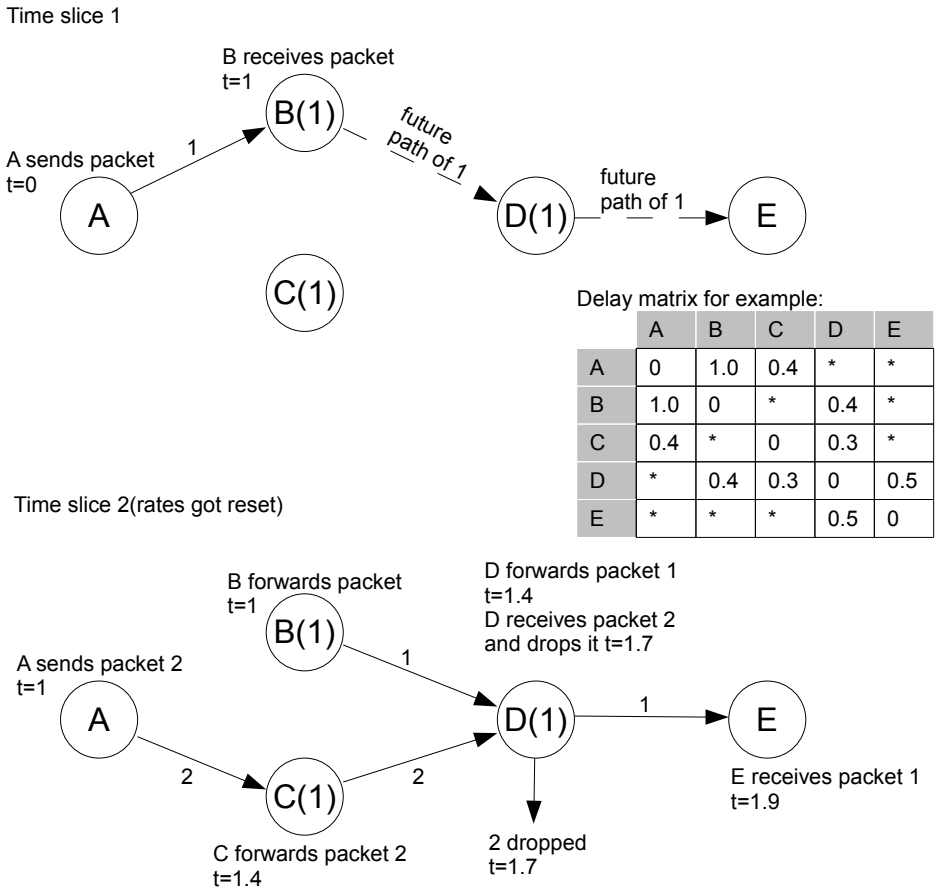
6.3.2. Perfect congestion controller

We assume we have a central manager that has all of the knowledge each node has and receives that knowledge instantaneously. This is, of course, something that we can do only in the simulation, but as the purpose of this congestion controller is to give us some upper limit a congestion controller can achieve, we can live with that.
At first glance the problem looks simple: Each node has some rate limit R_l for forwarding packets and it, of course, knows that rate limit, thus the manager knows it. That manager also knows the overall number of nodes N. Thus every node can use up $R_l/(N-1)$ (the node does not use itself) of the forwarding rate. This would be fair. But we have other problems:

1. If a node does not want to send something it will not use up its share. We could then distribute the rate that is left: If $R_p \leq R_l/(N-1)$ is the rate that was not used up each nodes would get $R_n + R_p/(N-2)$. But the problem is: When do we know how much was not used up? Ideally we would want to know that before each time slice. This would either need some prediction mechanism, which could be wrong and thereby disturb our 'perfect' congestion controller, or the application layer has to announce how much it wants to send in the next time slice and then only send this much. The second option would be feasible in our simulation, but would further complicate the application layer.

2. We did not think about round trip times. Packets sent along routes with long delays can collide with packets sent on routes with short delay (see figure 6.4).

3. If we only want to send packets with a specific size we have a problem: Consider $R_l/(N-1)$ to be smaller than that size. We would not be able to send a packet at all. The obvious solution is to split up that packet. This increases overhead and complicates the implementation and throughput measurements.



	A	B	C	D	E
A	0	1.0	0.4	*	*
B	1.0	0	*	0.4	*
C	0.4	*	0	0.3	*
D	*	0.4	0.3	0	0.5
E	*	*	*	0.5	0

Figure 6.4.: A sends packets to E using the intuitive version of the perfect congestion controller. Because of the latency of 1 between A and B packet 1 is longer than one time slice on the way and causes a packet sent in the next time slice to be dropped.

Instead of using the intuitive approach we propose following algorithm:
The manager is responsible for finding suitable paths. It returns the first one it finds. First all rate resets in all nodes are synchronized. That means the current rates of the nodes

R_c are set to zero at exactly the same moment. We can do that because we assume that central manager instance. The route finding in the manager works as in algorithm A.0.1 in the appendix. The algorithm looks through $\frac{(N-1)!}{(N-h-1)!}$ (h being the number of hops) possible paths and is thus quite inefficient, especially because it looks at all those possibilities every times a new path has to be selected. We can optimize that. If we cannot find any path from node n at time t_1 and at time t_2 with h hops and $|t_1 - t_2| < T_s$ with T_s the duration of a time slice, we can be certain that we will not find any paths for $t \in [t_1, t_2]$ with h hops either. See algorithm A.0.2 in the appendix for the optimized version.

6.4. Live video streaming

In this section we have a closer look at how the live video streaming application was implemented and what the general structure of the different programs (client and server) is. A detailed view at the tree construction algorithm, that is used to distribute the streaming data, is given as well.

6.4.1. General

The streaming server and client is implemented in C++ using Boost.Thread for maximal platform compatibility. As it was written for demonstration purposes it is only able to stream TS streams as defined in ISO 13818-1. The implementation is able to be plugged in into existing streaming architectures as described in Figure 6.5. The streaming is done

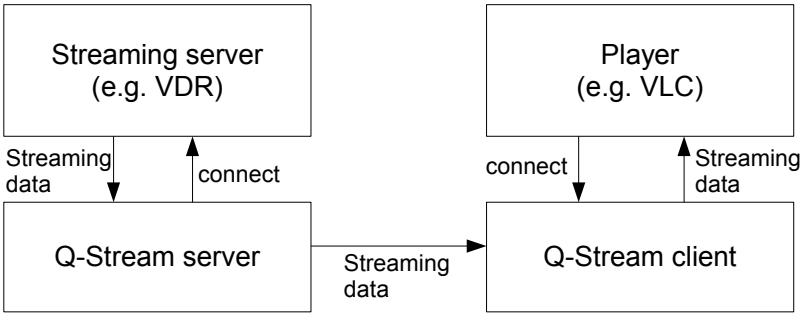


Figure 6.5.: The Q-Stream server connects itself with the real streaming server using HTTP. It then receives the stream which it forwards to its clients. Those open a HTTP port as well to which client players can connect and receive the stream.

using following steps:

1. When the Q-Stream server is started a parameter is given pointing to a real streaming server (which gets its data via DVB-S/T/C or by other means). Then the Q-Stream server connects itself to that server using a HTTP requests. It then receives the stream which it distributes using the above mentioned technique.

2. Each packet the server receives gets an unique id. The Q-Stream clients caches out of order deliveries using this id. If the next part of the stream is available it is forwarded to a player application. See Figure 6.6. Actually it is way more complicated, because the client tries to only forward TS packets, which are 188 bytes long, to the player.

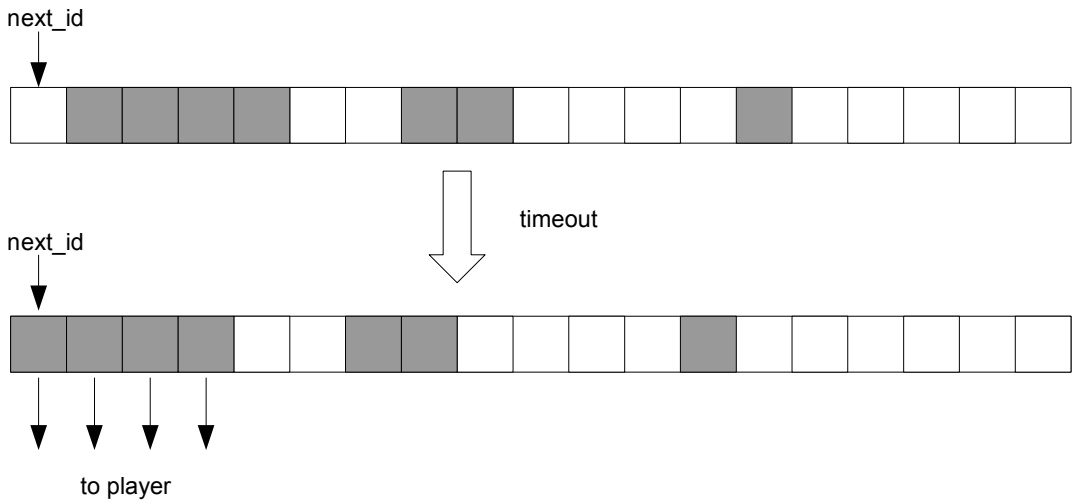


Figure 6.6.: The queue in a Q-Stream client. *next_id* is the id of the packet the client needs next. The gray fields indicate packets that have already been received. If it does not get this packet in time a timeout occurs and *next_id* is incremented. If *next_id* points to an available packet it is send to the player at once.

3. The client is able to forward TS packets to the player by opening a listening HTTP-port where this player application can connect to. This way the Q-Stream server and client build a tunnel. Ideally the stream the player application gets from the Q-Stream client is exactly the same one as if it requested the same stream from the real streaming server.

6.4.2. Q-Stream server

The Q-Stream server is implemented with three threads, each having a unique set of tasks:

1. **Input thread**
   The input thread connects itself to the real streaming server and then receives the streaming data which it writes into a buffer that can be accessed from the other threads.

2. **Tracker thread**
   The tracker thread opens a TCP port to which the clients initially connect to and announce their ports on which they want to receive the UDP streaming packets. The tracker thread regularly sends pings to its clients. If a ping is not received in time the client is assumed to be dead and is disconnected. The tracker thread receives positive and negative acknowledgments from its clients, which other threads can access. It gets information about client throughput rates from the controller thread and uses this information to construct trees to optimally distribute the stream.

3. **Controller thread**
   The controller thread does the exploration, handles the congestion control using acknowledgments it gets from the tracker thread and uses the trees constructed in the tracker thread to distribute the packets it gets from the input thread.

6.4.3. Q-Stream client

The Q-Stream client is implemented using three threads as well. Those are

1. **Controller thread**
   This thread receives exploration and exploitation packets from other clients or from the streaming server via UDP and forwards them. Exploration packets are sent to the next client in the exploration chain or, if the current client is the last one, an acknowledgment is send to the server via TCP. Exploitation packets are send via UDP to all children the clients have in the respective tree.

2. **Tracker connector thread**
   Thread that connects itself to the tracker. Announces the port the client listens for UDP packets and the bandwidth it thinks it is able to handle. Receives the children of each client for different stream slices and responds to pings.

3. **Output thread**
   Listens on a TCP socket and sends the video stream it gets from the controller thread to all applications who connect to it via HTTP.

6.4.4. Tree construction

There are k different slices (k can be arbitrarily set. Bigger k increase the administrative costs). For each slice there is a different tree. Because depending on the number of clients and the bandwidths these clients are able to forward, the number of possible trees exponentially increases and then gets multiplied by k, as we have k different trees, doing the tree construction is possibly very costly. There is no intuitive, non NP algorithm which constructs a tree with optimal structure (fulfill bandwidth constraints and minimize latencies between the nodes). Because the latencies between nodes and their bandwidths change continuously and we want to change the tree as little as possible as well, an iterative approach is taken. Regularly a little optimization by exchanging nodes or attaching a node to a different parent if this increases the overall tree performance is done.

If a new client joins the streaming server it is added as leaf node to a node that can handle this new node. This is done in each of the k trees separately. If this is not possible the client gets a message that the streaming service is overloaded. The node which is now the parent of the new client is informed about that. The streaming server checks repeatedly if the current tree structure is optimal. This is measured by the sum over all latencies between the clients and the streaming server in the trees(One could further introduce some value for reliability, but this is not done yet) as in Figure 6.7. Lower performance values are better. A tree is optimized by using two operations:

1. **Adopt** Each node tries to adopt each other node by making it its child. If this is possible (the node has enough bandwidth available) the performance of the resulting tree is compared to the performance of the current tree (using the sum over all latencies). If the resulting tree is better it is used from now on and all involved clients are informed of that change. This decreases the depth of the tree, because the performance metric used will result in nodes higher in the tree hierarchy adopting nodes lower in the hierarchy.

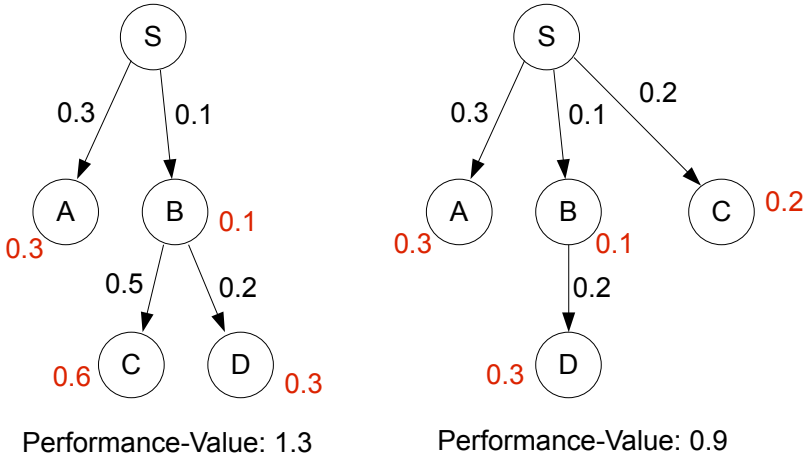


Figure 6.7.: The performance of a tree is defined as the sum over all latencies from the clients to the streaming server. The latencies to the streaming server are the values on the nodes. The edge weights are the latencies between the nodes. This metric favors trees with small depth and low latency between the nodes.

2. **Switch** Each node tries to switch its place with each other node. If both nodes are able to handle each other's children and the performance of the resulting tree is better, the switch is done. This results in trees with lower latency, because nodes with low latencies to another's children will exchange their place. Likewise if a node has no children, is able to handle the children of another node and has lower latency to them than the current node, they switch their places. All nodes who then have different children are informed of that change.

If a client leaves, all the orphans it leaves behind are assigned again as leaf nodes to the tree like if a new node joins (only now this node can already have children). If the number of children a client can handle is lowered by the congestion control algorithm beyond the number of current children, these children are orphaned and reassigned as well.
The three operations Join, Switch and Adopt can also be seen at Figure 6.8.

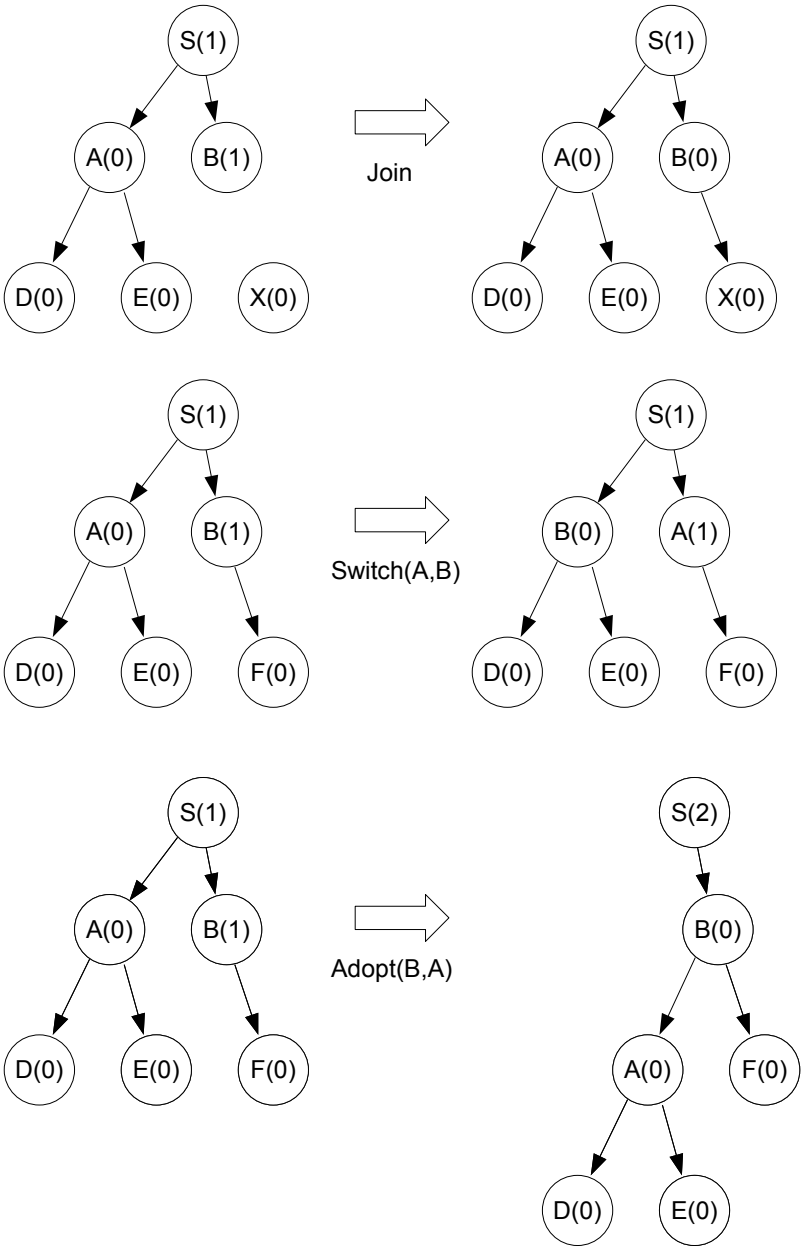


Figure 6.8.: The three operations the streaming server does to optimize the tree. The number in the brackets indicate how many more children this node is able to handle. S is the streaming server.

Chapter 7.
Results

In this chapter, the performance of the different congestion controllers is analyzed. To ensure comparability the parameters of the two existing approaches are optimized. Then the performance of the two approaches is compared in several benchmark setups and scenarios. It is also shown that the Q-Learning congestion controller can be successfully used to distribute a live video stream.

7.1. Parameter optimization for the AIMD approach

Before comparing the different approaches we want to look at how different values for different parameters affect the congestion controllers and then make an informed decision about which values we take.

7.1.1. Parameters

Used by the round trip time estimation:

- γ in equation 4.3 and 4.4
- α in equation 4.5 and 4.6

Used by the update rule:

- α in equation 4.7
- β int equation 4.11

As already discussed the two parameters in 4.4 and 4.6 (the RTT estimation) have to be set to $\alpha = 1$ and $\beta = \frac{1}{2}$ if we want to be fair to TCP, but to compare the AIMD approach to the Q-Learning approach we have to select the optimal ones as well.

7.1.2. Goal

We have two conflicting goals:

1. We want to have as much throughput as possible
2. We want to have as few packets dropped as possible

Why conflicting? To have the optimal throughput we have to test regularly if we can increase rates. If we do that we can cause packets to be dropped. The optimal balance is dependent on the specific problem (available bandwidth, structure of the network,....), so we can only establish some guide values here.
While optimizing the parameters for the round trip time estimation we do have to deal with conflicting goals as well. But those can be seen independent of the above goals and thus we can tackle the problems separately.

7.1.3. Testbed

The parameter selection was always done with 20 nodes, simulating for 10000 seconds using 400000 bytes/s available bandwidth at each peer. Packets are always sent over three hops. Delays between nodes are calculated using a distance metric using the nodes' positions. The nodes are assigned the same positions each run. The application sends packets until all peers are saturated.

7.1.4. Results for RTT paramters

The time for convergence with different γ and *alpha* is shown in table 7.1(top left and bottom). The time measured here is the time until the difference of the average time to a timeout (we receive an ACK and compare it to the time it was predicted to be received) over one second and the average time to a timeout of the previous second is smaller than some difference (here $1e-5$). The jerkiness of the curve may be resulted by the low repeat count of 3, but the simulations ran long enough as it is. One typical run of the course of the time to a timeout can be seen in figure 7.1. The number of delayed acknowledgments (acknowledgments that were received after the acknowledgment timeout occurred) can be seen as measurement for the quality that needs to be balanced with convergence speed. This can be seen at table 7.1(top right and bottom right). If the time to timeouts is large, there should be less delayed acknowledgments. If we smooth the curves (extensively) we can see that increasing γ beyond 0.15 gains only little advantage with regard to convergence speed, while the number of delayed acknowledgments steadily increases. Therefore, if convergence speed is needed, $\gamma = 0.15$ should be chosen. For α 4 seems to be a good value.

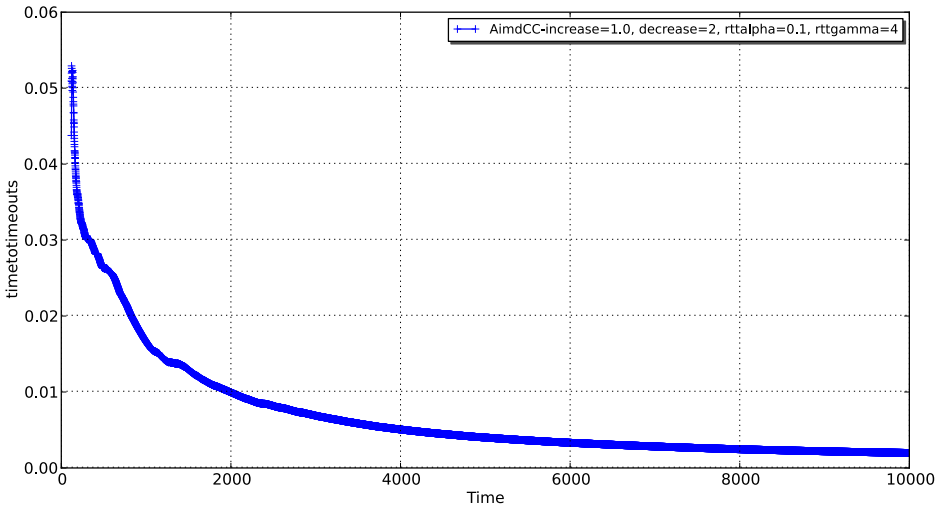


Figure 7.1.: Typical curve for the time between receiving an ACK and the time it was predicted to be received averaged (arithmetically) over one second. With $\gamma = 0.1$

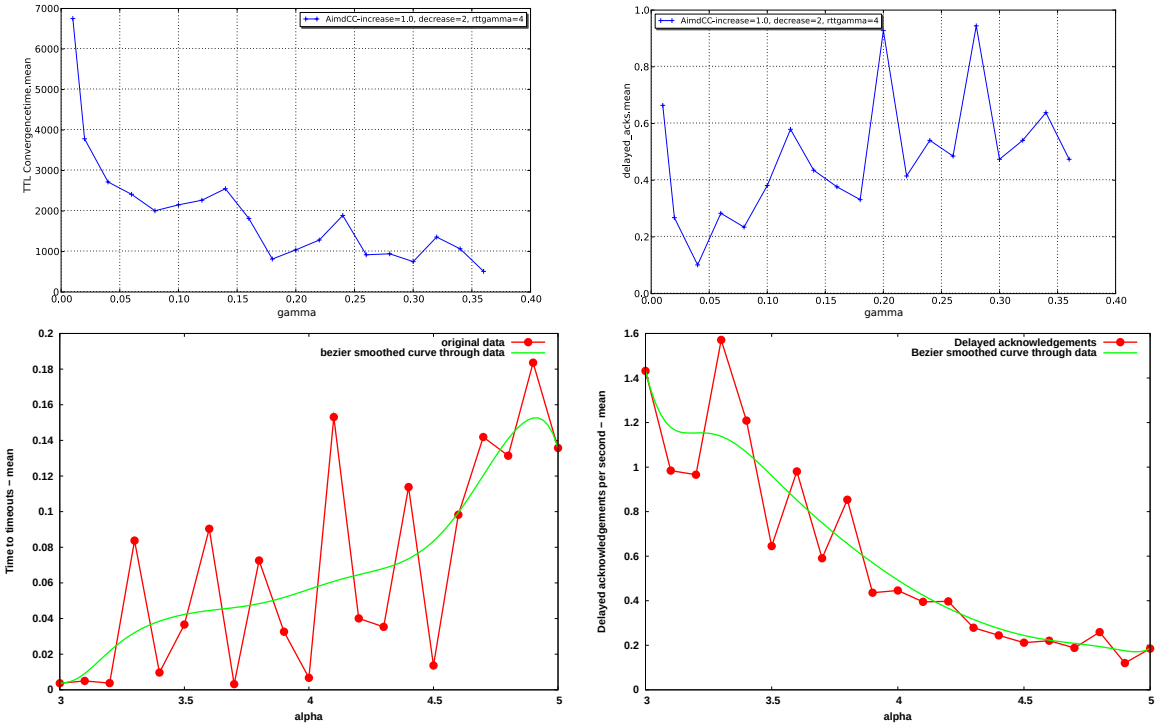


Table 7.1.: Top left: Time till convergence of the time to the timeouts. Top right: Arithmetic average of the delayed ACKs per second measured after convergence. Bottom left: Arithmetic average of the time between receiving an ACK and the time it was predicted to be received measured after convergence. Bottom right: Arithmetic average of the delayed ACKs per second measured after convergence. Each of the points in the top plots is the average over three simulations of 20 nodes for 10000 seconds with parameter γ and each of the points in the bottom plots is the average over three simulations of 20 nodes for 10000 seconds with parameter α.

7.1.5. Results for the update rule parameters

We have three dimensions we want to optimize the update rule parameters for: The throughput of the whole system, the number of dropped packets caused by congestion and convergence speed. The parameters have direct influence on the convergence speed. α in equation 4.7 sets how many MTU-sized packets should be added per round trip time interval. It is logical that higher α increase convergence speed while lower α decrease it. Because measuring convergence speed is difficult and selecting the right parameter for fast convergence depends mainly on the use case, we just optimized the other output dimensions.
In Fig. 7.2 one can see the mean throughput of the whole system with different α and β values. Together with Fig. 7.3, which shows the mean drops for different parameters, we can see that a selection of $\alpha = 1.5$ and $\beta = 1.4$ is reasonable. Using these parameters there are relatively few drops for a high amount of throughput.

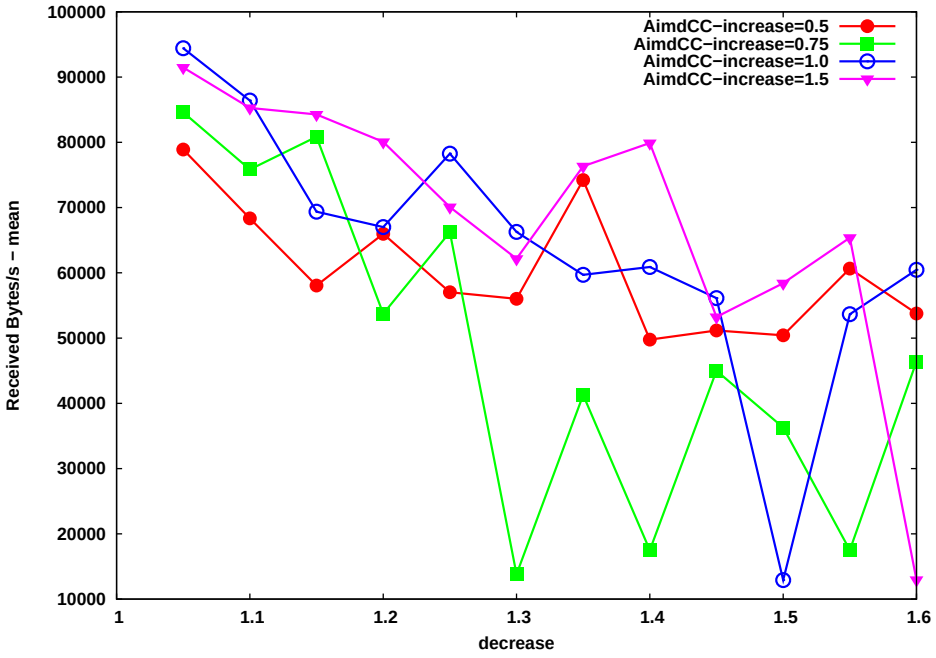


Figure 7.2.: Arithmetic average of the added throughput of the whole system per second. Measured after convergence. Each measurement point in the plot is the result of one simulation with 20 nodes for 10000 seconds with parameter α(increase) and four different β(decrease).

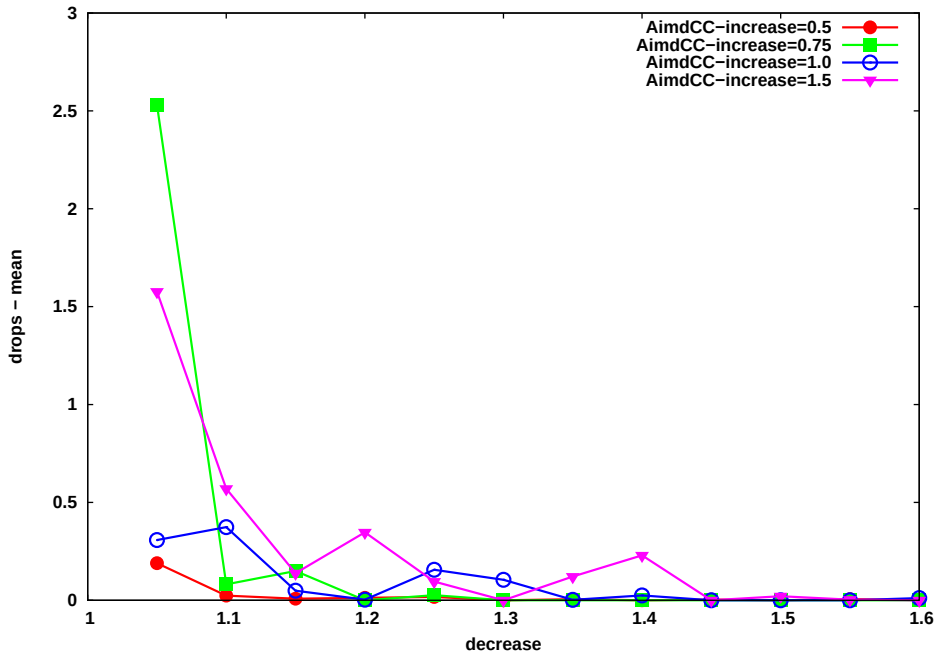


Figure 7.3.: Arithmetic average of the dropped packets per second. Measured after convergence. Each measurement point in the plot is the result of one simulation with 20 nodes and for 10000 seconds with parameter α(increase) and four different β(decrease).

7.2. Parameter optimization for the Q-Learning approach

As we use the same round trip time estimation algorithm for the Q-Learning congestion controller as for the AIMD-controller, we can reuse the results from the above section with regard to round trip time estimation. Not withstanding, we do have more free parameters in the Q-Learning approach than in the AIMD approach. Those are optimized here by making strong assumptions about their independence.

7.2.1. Free parameters

We have:

1. In equation 5.4: α and γ
2. Rewards for acknowledgments and acknowledgment timeouts: r_u(ACK) and r_d(NACK)
3. The percentage of random actions in the ϵ-greedy exploration strategy: ϵ (See section 5.4.5)

7.2.2. Independence assumptions

We do not optimize parameter ϵ at all. The value this parameter should have depends too much on the network topology and use case, because it sets how fast the Q-Congestion

controller will adapt to changes in its environment. If the bandwidth a Q-Congestion controlled connection has changes often we want to have a bigger ϵ, as then the bandwidth is adapted and utilized faster. On networks with low bandwidth changes we like to have a small ϵ as the smaller the ϵ the more efficient the equilibrium. This is the exploration/exploitation dilemma also mentioned in section 5.3.2.
Parameters α, γ, r_u and r_d should depend on each other, but it is assumed that the dependence of α and γ on r_u and r_d is small. The rationale behind this is that r_u and r_d mostly influence how efficient the used rates at the equilibrium point are and convergence speed only to a lesser degree, whereas α and γ mostly influence convergence and adaption speed and not as much the equilibrium point, because at this point the Q-Values should be subject to only few changes.

7.2.3. Testbed

The testbed is the same as in the AIMD experiments. See section 7.1.3.

7.2.4. Results

We do not need both r_u and r_d to steer the balance at the equilibrium point. Because of that only r_u is optimized and r_d is set to -1.0. In table 7.2 we see that the optimal r_u for 20 nodes with 40000 bytes per second incoming bandwidth for the overlay is approximately $r_u = 0.015$, if we can live with a higher drop rate. Otherwise we should select $r_u = 0.01$.

In table 7.3 we see that a reasonable value for α is 0.25 and for γ 0.9. If we want few drops $\alpha = 0.2$ would be a better choice.

7.3. Artificial Benchmarks

7.3.1. Saturated Connections

We simulate with saturated connections (the peers send as much as they can), 20 nodes for 1000 seconds up to 300000 seconds depending on how much data we need to get Gaussian distributed means. The individual nodes forward up to 400000 bytes per second overlay data. Each packet is routed over exactly three hops. Each packet is sent to a random other node. The optimized parameters were taken for all methods. After the simulations arithmetic means of the measurement data is taken, so that we have approximately 35 data points for each measured feature. We look if we can assume these data points to be normal distributed by a quantile-quantile plot. Autocorrelation is investigated as well. We can then calculate confidence intervals (here 95%) assuming that the data points are normal distributed. For distance and latency simulation the coordinate based delay build into Oversim[38] with a jitter of 10% was used.
Table 7.5 shows an example how a rate from one peer to another can look like in this scenario using the different approaches.

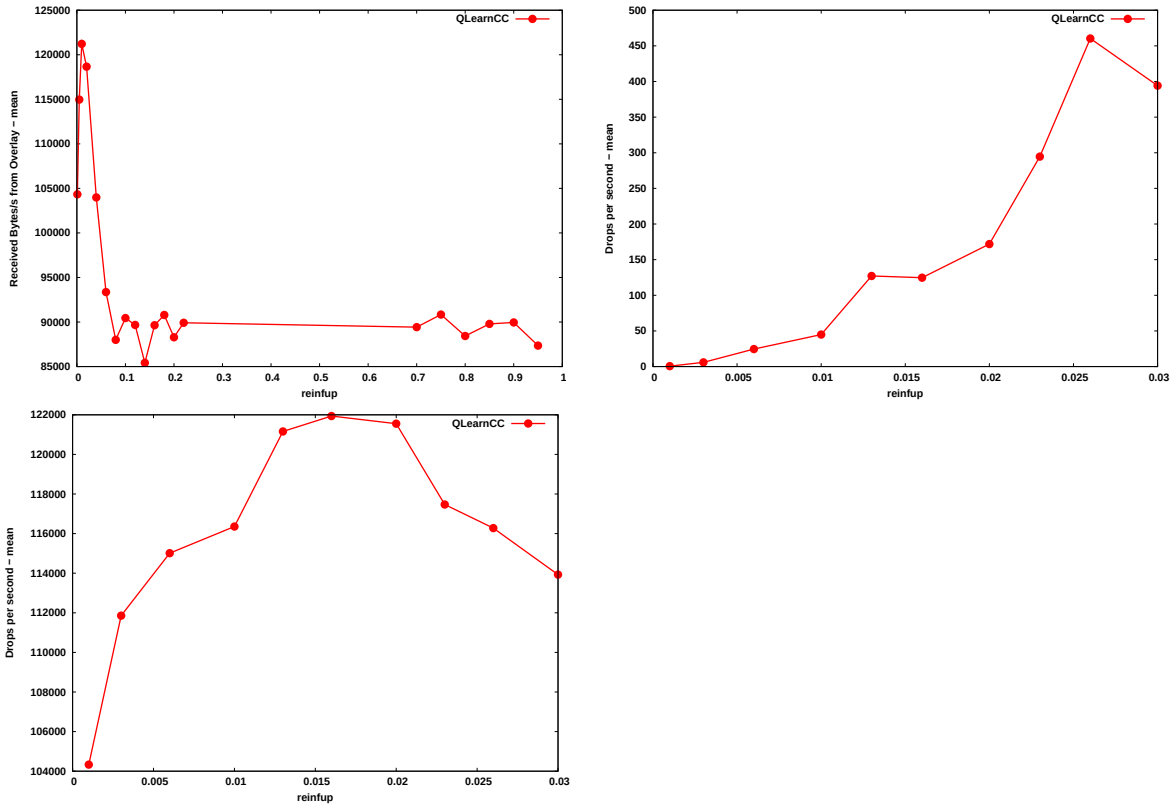


Table 7.2.: Top left: Mean received data per second with variable r_u. Bottom left: Scaled version of the top image for the range $r_u = [0.001, 0.03]$. Top right: Mean packet drops per second with variable r_u for the same scale as the bottom left picture. Each measurement point in the plot is the result of one simulation with 20 nodes for 10000 seconds.

Results

As one can see in table 7.4 the perfect congestion controller is slightly worse than the Q-Learning controller in this scenario. The reason is it had less time to estimate the round trip times (because of computational reasons the perfect controller ran for 1000 seconds and Q-Controller for 300000 seconds). These imperfect estimations cause the higher drop rate as well. In Figure 7.4 the perfect controller is better because there the Q-Congestion controller runs only for 10000 seconds.
The convergence of the Q-Learning approach is better. This means the fast-start mechanism works better there.
The random 'controller' is absolutely useless, as the drop rate is very high and the throughput very low.

Figures 7.4-7.8 and Table 7.6 show results for the same scenario as used for Table 7.4 (20 peers forwarding up to 400000 bytes per second overlay data, three hops, coordinate based delay with a jitter of 10%). One of these scenario parameters was varied in each plot.

[1]Mean bytes per second with 95% confidence interval. See tables A.1-A.4 for quantile-quantile and auto-correlation plots.
[2]Time till the average throughput over 100s is the first time greater than 90% of the mean throughput.
[3]In percent of the total received messages and over the course of the whole simulation.

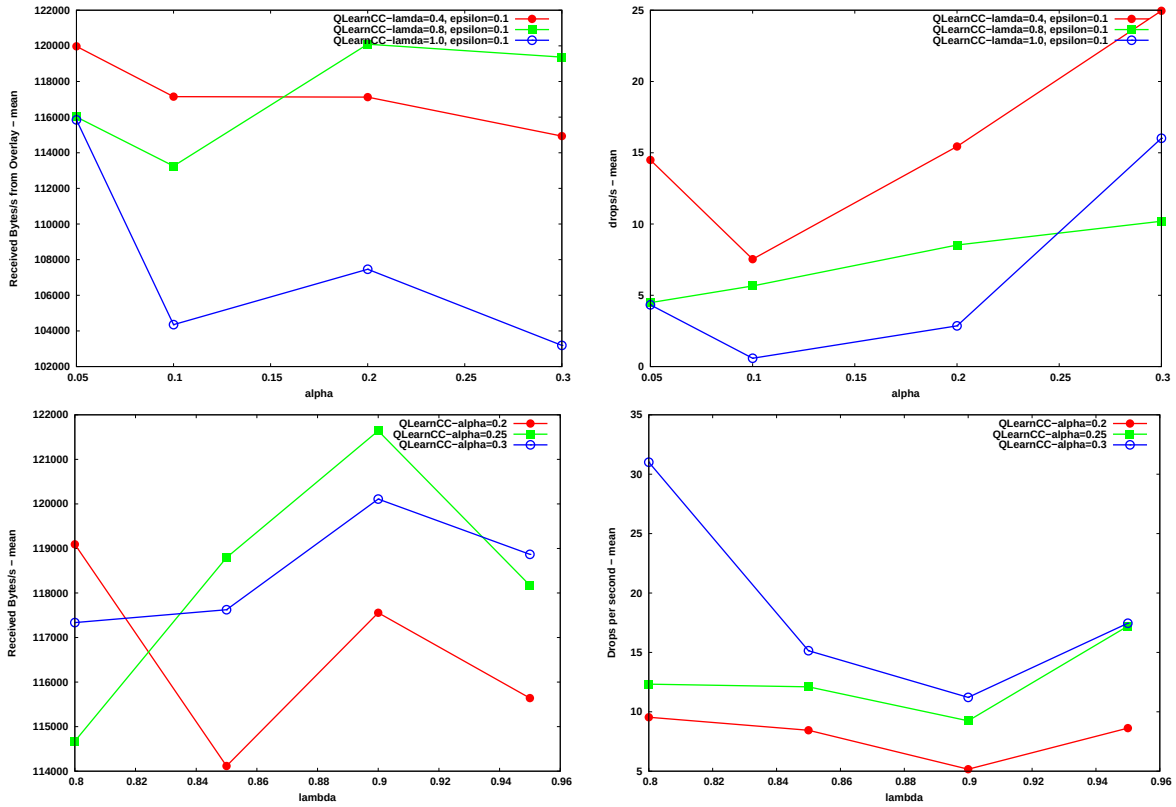


Table 7.3.: On the left: Mean received data per second with variables α and γ(here inadvertently called lamda or lamda) (see eq. 5.4) with different scales. On the right: Mean drops per second with variables α and γ. Each measurement point in the plot is the result of one simulation with 20 nodes for 10000 seconds.

Figure 7.4 shows how the mean throughput of the different approaches behaves if the number of participating peers is increased. One can see that the performance of the AIMD approach gets better while the Q-Learn approach gets worse with an increasing number of peers. The reason for that is that with an increasing number of peers it becomes more difficult to learn the variables of other peers, because less information about them is available. Fewer packets are sent through a single peer, because the number of sent packets stays the same while the number of peers increases.
The AIMD approach with its severe reactions to congestion becomes better with an increasing number of peers, because the congestion reaction throttles the overall bandwidth less the more peers there are.

Figure 7.5 shows that the Q-Learn and AIMD congestion controller are next to unaffected by increasing the number of hops per packet. Their relative bandwidth usage compared to the perfect congestion controller stays the same. This justifies that an acknowledgment timeout for one packet is used as congestion signal for all nodes that packet was routed through (See section 4.2.1). If this figure showed that the mean throughput decreases more in the Q-Learn and AIMD congestion controllers than in the perfect congestion controller, this technique would have to be reconsidered.

Figure 7.6 shows that the Q-Learn congestion controller is nearly unaffected by higher delays between the nodes. The AIMD congestion controller (like TCP) behaves poorly with increasing round trip times. After increasing the latency between nodes to $200ms$

Approach	Throughput[1]	Time till convergence[2]	Mean drop rate[3]
AIMD	46642.57 ± 14.19111	$1368.5s$	$0.000275\% \pm 1.970 * 10^{-6}\%$
Q-Learning	90416.43 ± 24.46569	$293.5s$	$0.060195\% \pm 0.000635\%$
Perfect	89271.82 ± 6.599807	$0s$	$0.292215\% \pm 0.00125\%$
Random	4099.495 ± 4.823612	$0s$	$67.02509\% \pm 0.000654\%$

Table 7.4.: Simulation results for 20 peers with each peer forwarding up to 400000 bytes per second overlay data and three hops for each packet

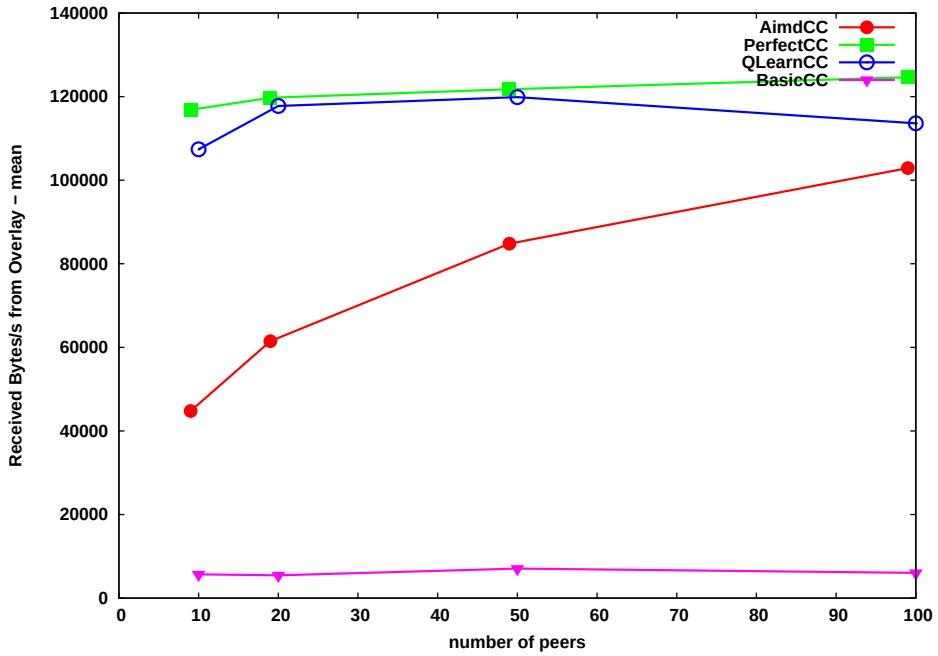


Figure 7.4.: Mean received bytes per second plotted against the number of nodes in the simulation.

it is barely better than the 'random' congestion controller. The reason for this can be seen at Figure 7.7. The lower the latency the more often an additive increase happens. Thus a higher latency causes higher recovery times from packet loss, and thus a lower performance.

Figure 7.8 shows what happens if the bandwidths of the peers are modified. Here the opposite effect of Figure 7.4 occurs. The higher the bandwidths the more information the Q-Congestion controller can gather and the better it is. The AIMD congestion controller becomes more attractive if less information is available. It cannot use the information that is there at higher rates. The Q-Congestion Controller can use this information and because of that has better performance.

Table 7.6 shows how the number of drops per second varies in different scenarios with the AIMD and Q-Learning approach. We see that with an increasing number of peers the Q-Learning controller causes lots of packet drops. We also see that the AIMD controller has next to no drops in most of the scenarios. This may be the result of the severe reaction (decreasing the rate multiplicatively) this approach has to congestion. We saw in Figure 7.6 that higher latencies between peers cause the AIMD controller to increase individual rates slower. In Table 7.6 we see that the drop rate then goes down to zero with increasing

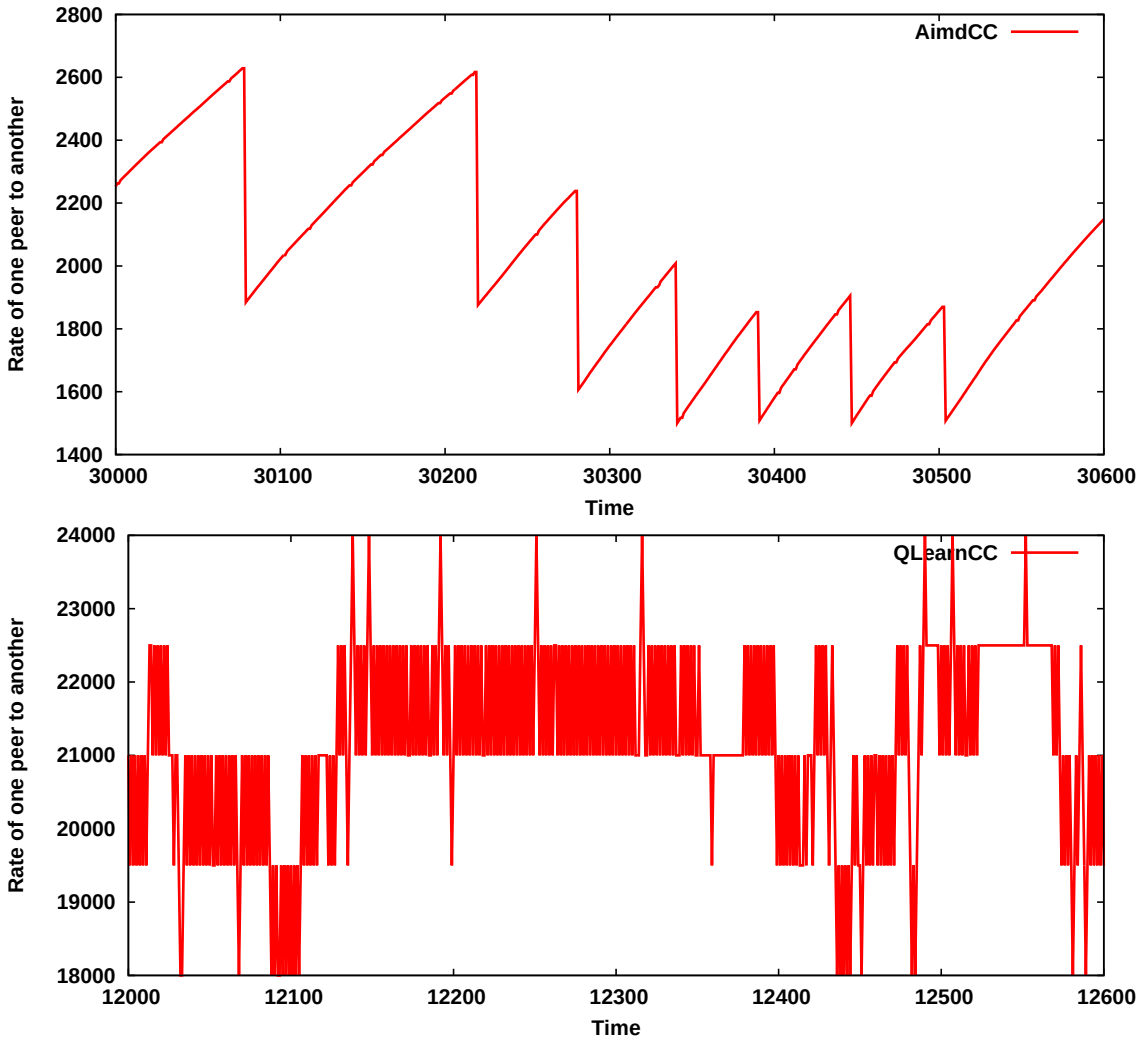


Table 7.5.: A progression of rates for 600 seconds from one client to one other using the AIMD and Q-Learning approach.

latencies: The rates are growing so slow and are thereby that small, that no drops occur. The Q-Learning controller has a lot more drops per second in most scenarios.

7.3.2. Rapidly changing traffic

We use the same scenario (20 peers forwarding up to 400000 bytes per second overlay data, three hops, coordinate based delay with a jitter of 10%) to see how the approaches can cope with rapidly changing traffic from peers. Something like that can occur if new peers join the network (churn) or the applications using the overlay network sometimes do not use all available outgoing bandwidth and then suddenly use more of it (If the application allows the peers to download files this would occur if the user starts or stops downloads). To see how the different approaches cope following experiment is used: At the beginning none of the peers send anything. Every 500 seconds one of the peers that has previously sent nothing starts sending until saturation (as much as the congestion controll algorithm allows it to send). Each peer that started sending until saturation does not stop again. We simulate for 10000 seconds. At the end all peers should have started sending until

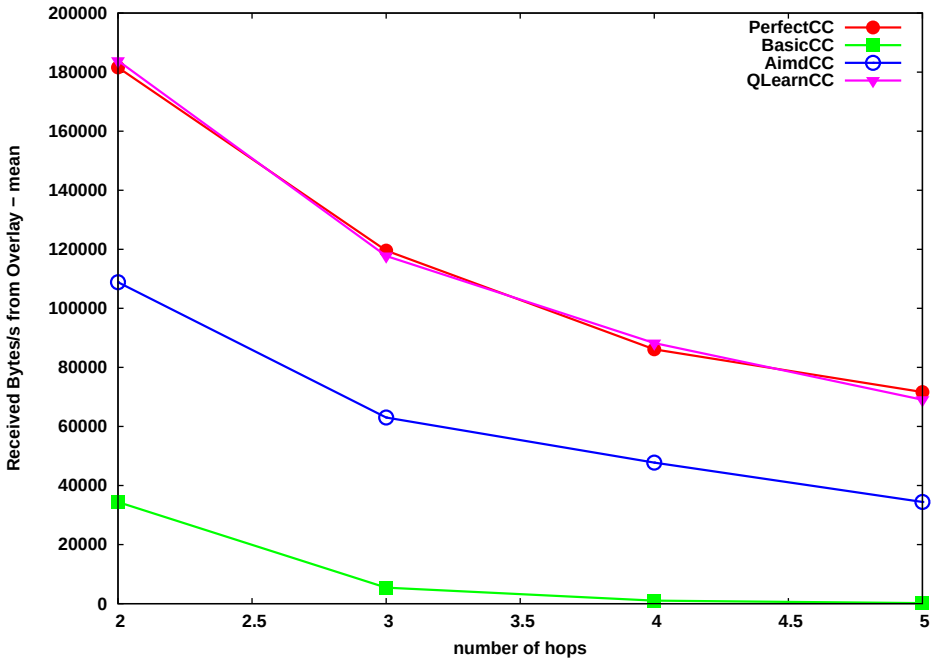


Figure 7.5.: Mean received bytes per second plotted against the number of hops for each packet.

saturation.
Table 7.7 shows how the different approaches fare with regard to the throughput of the network and the drops (congestions) that occur. We can see that the Q-Learning approach does better than the AIMD approach, but causes a lot of drops to occur and is not quite as good as the perfect controller. The AIMD controller seems to do bad because of the slow start mechanism. The peer that starts sending is at slow start state at the beginning until the first acknowledgment timeout occurs. This causes the spikes in throughput and drops. The slow start mechanism not only causes drops for that starting peer but for everyone else. Because of the overusage of some nodes due to the slow start mechanism several peers have to adjust their rates multiplicatively, which decreases throughput that much that it does not recover within the 500 seconds at which the next peer starts sending and causes drops again.
One can see in the upper graph in Table 7.7 a 'step' like behavior with the Q-Learning and perfect congestion controller, when the first few peers start sending. This is not caused by the congestion controllers, but an implementation issue. The connections of the first few sending peers could not be saturated. The curve of the perfect controller shows, however, that after four peers started sending, they all can reach saturation.

7.3.3. Adaption speed to changing bandwidth

It is interesting to know how fast the two approaches adapt to changed client bandwidths. Such a change could occur if other applications that use the same medium become active or are finished or the client changes the bandwidth it has allowed the overlay to use. To test how the AIMD and Q-Learning approaches react in such a scenario following experiment has been devised:
There are 20 nodes, 10 of them allow the overlay to use 400000 bytes per second of outgoing and incoming bandwidth. The other ten only allow half of it: 200000 bytes per second. The

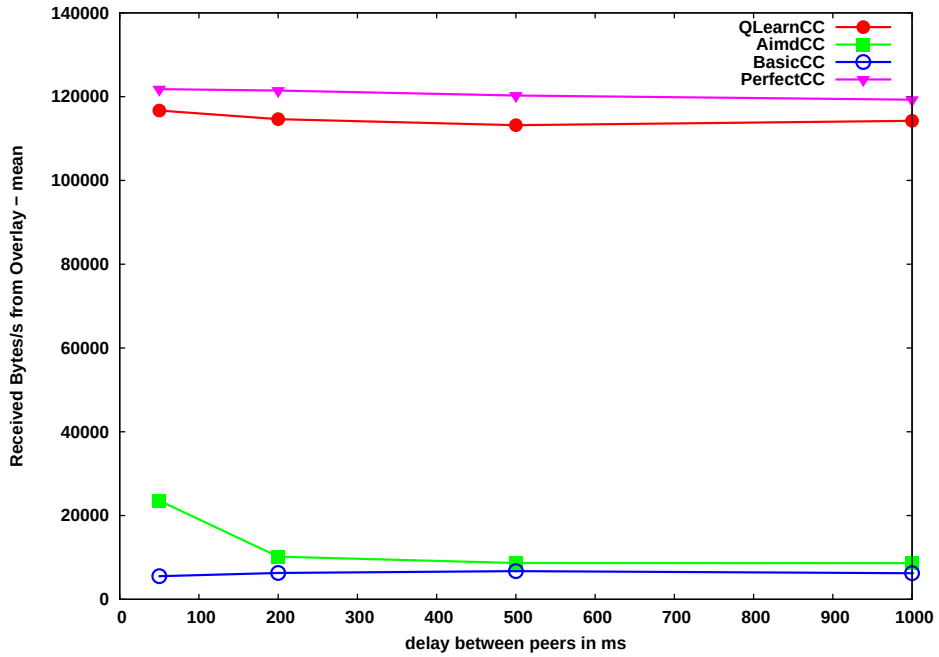


Figure 7.6.: Mean received bytes per second plotted against the delay between peers in milliseconds. For this plot the coordinate based delay mechanism of Oversim was disabled and a constant delay was used (with the same jitter percentage of 10%).

simulation runs for 20000 seconds. After 10000s the peers allowing 400000 bytes per second suddenly allow only 200000 bytes per second. The other half suddenly allows 400000 bytes per second. It should be clear that after that exchange the throughput of the whole system should be the same as before the exchange.
The compared throughput of the overlay is shown in figure 7.9. One can see that even in this long time (10000s) the Q-Learning approach does not reach the throughput it had before the exchange. It seems it is difficult for the Q-Learning controller to adapt to a changed environment. The reason for this is clear: The Q-Learning controller learns about the rates of the clients and it is painfully slow to change that learned behavior. The very thing that makes this controller good at equilibrium points is now a disadvantage: If congestion occurs it does not immediately decrease its rate to the respective client, but adds it to the experience it previously had. This makes the controller more resistant to random packet loss and able to reach such an efficient equilibrium. Likewise if it has learned which state is optimal only the ϵ-greedy action selection (see section 5.4.5) prevents that it will never leave this state. This means that the realization that there is a better state only comes with probability ϵ (Increasing this parameter would improve the responsiveness in this experiment, but also lead to less efficient equilibrium throughputs). Thus it learns even slower that higher bandwidths are possible than it learns about packet losses with current bandwidth usage. This can also be seen in figure 7.10 where after about 250 seconds after the exchange the drop rates previous to the exchange are reached, whereas even after 10000 seconds previous throughput cannot be obtained.

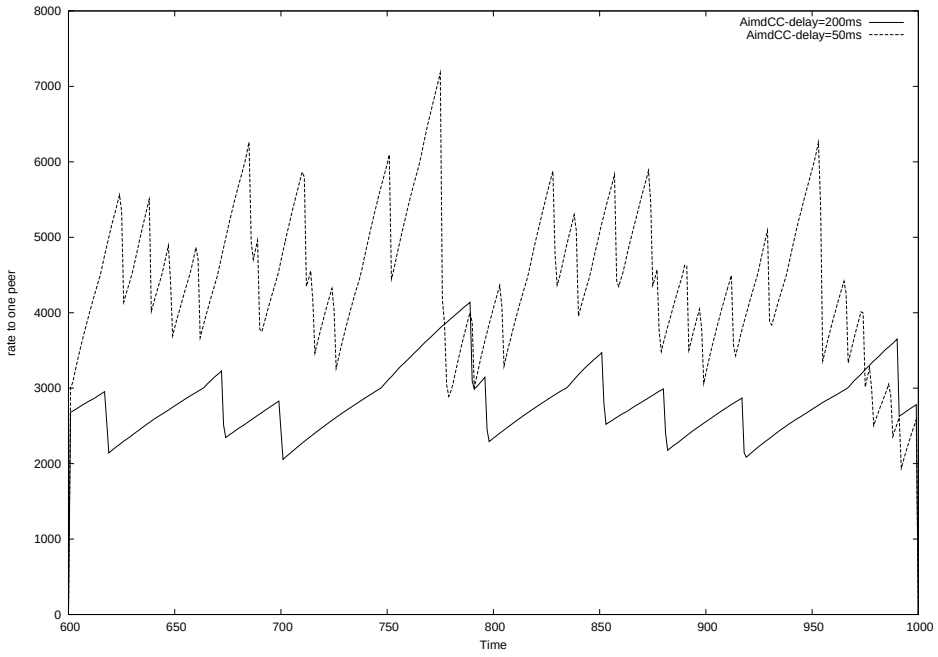


Figure 7.7.: Rates from one peer to another in the AIMD approach with different latencies between the clients.

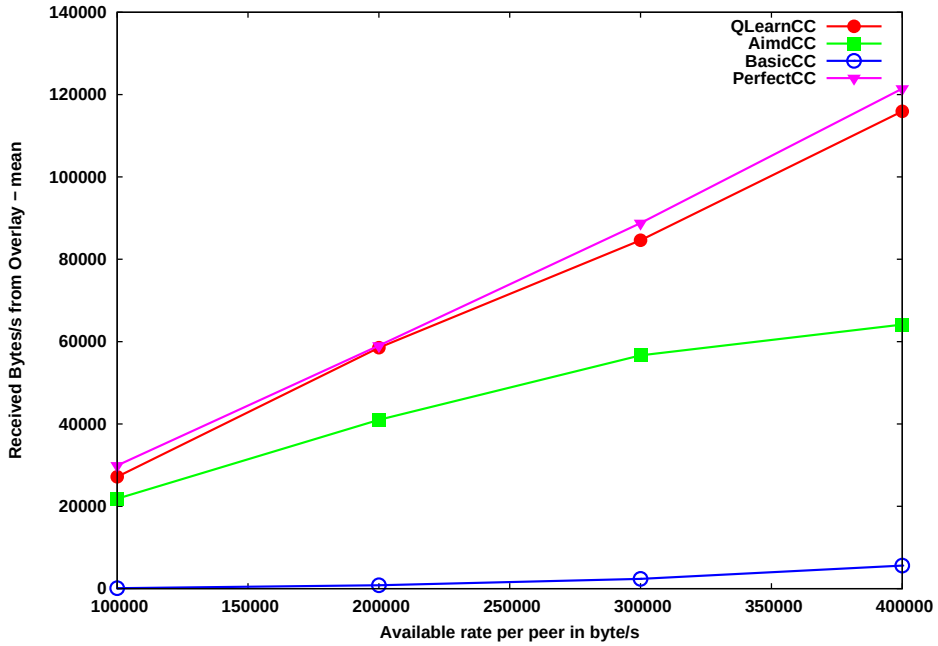


Figure 7.8.: Mean received bytes per second plotted against the available rate at each peer in bytes/s.

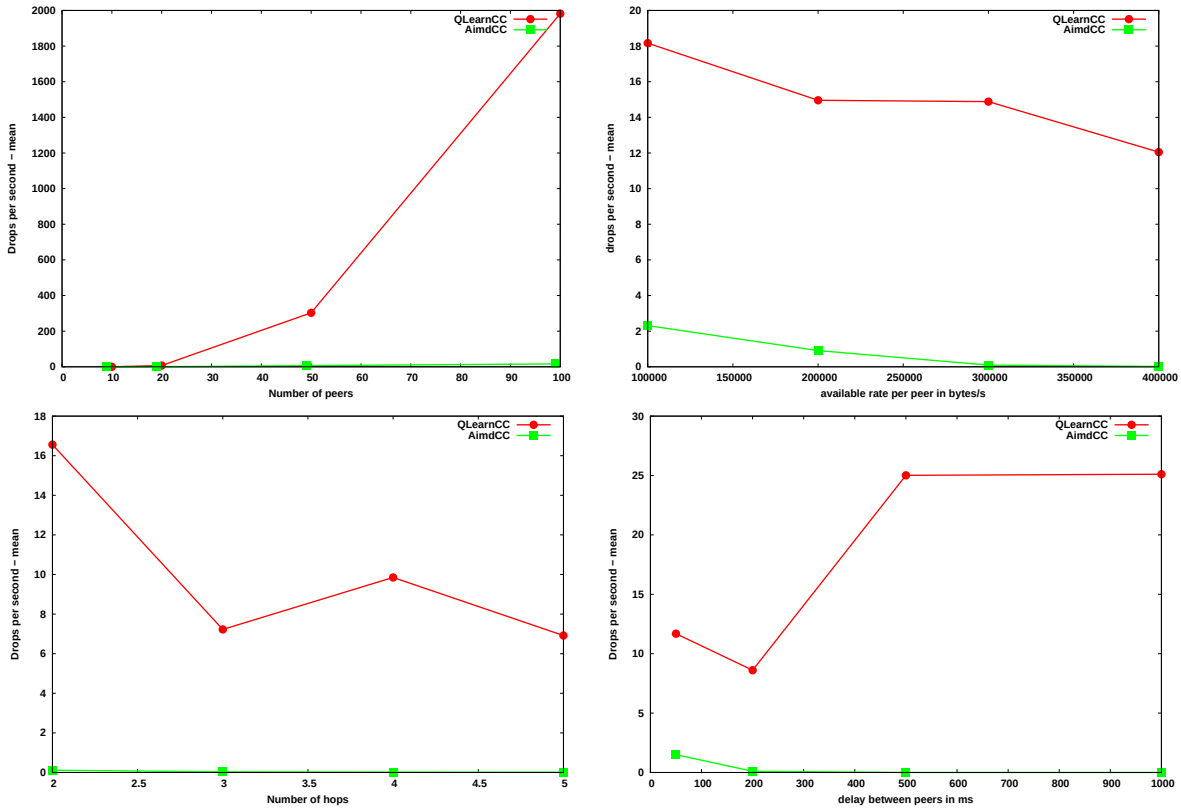


Table 7.6.: Drops per second plotted against the number of peers in the simulation, rates each peer is able to handle, number of hops each packet has to take and the latency between the peers.

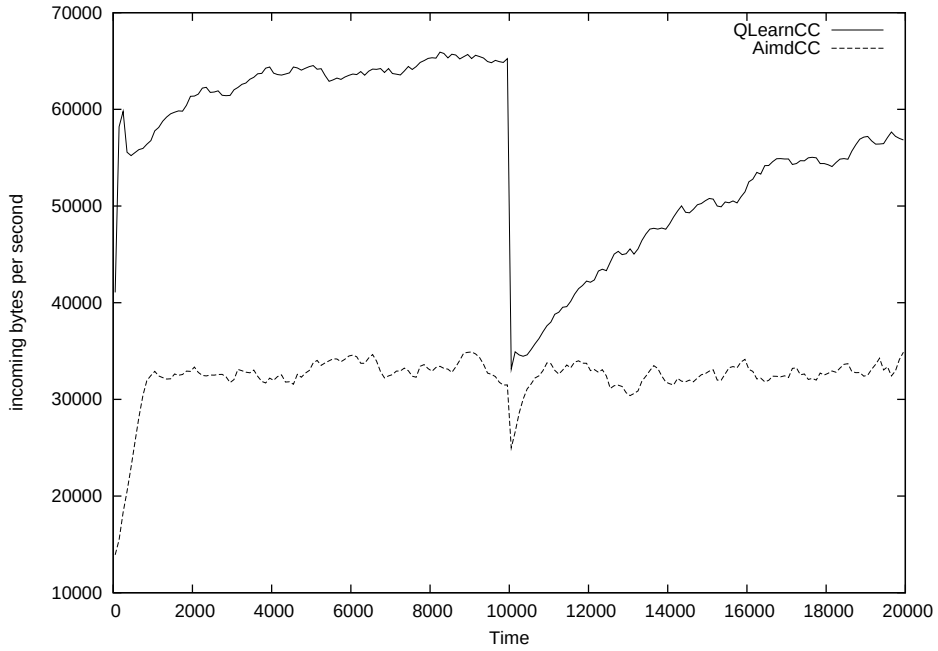


Figure 7.9.: Graph comparing the AIMD and Q-Learning approach. After half the simulation the rate limits of the peers are exchanged. The throughput should reach the value before the exchange. This happens very fast in the AIMD approach. The Q-Learning approach fails to do this.

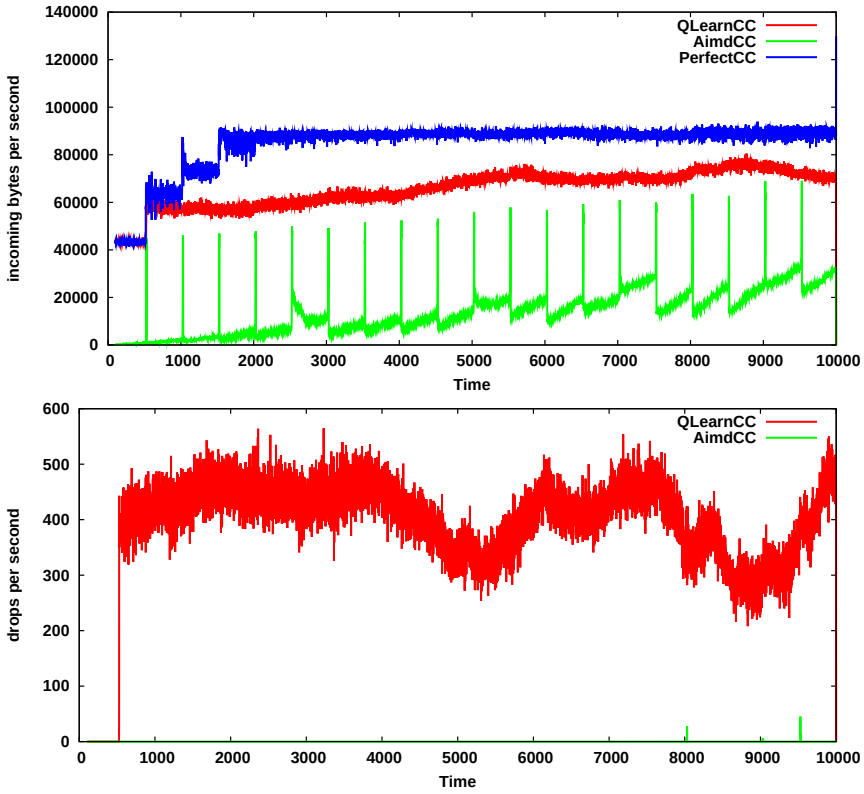


Table 7.7.: The AIMD and Q-Learning approach compared to the 'Perfect' congestion controller. Every 500 seconds a peer starts sending until saturation. The Q-Learning controller has better performance at the cost of a lot of drops.

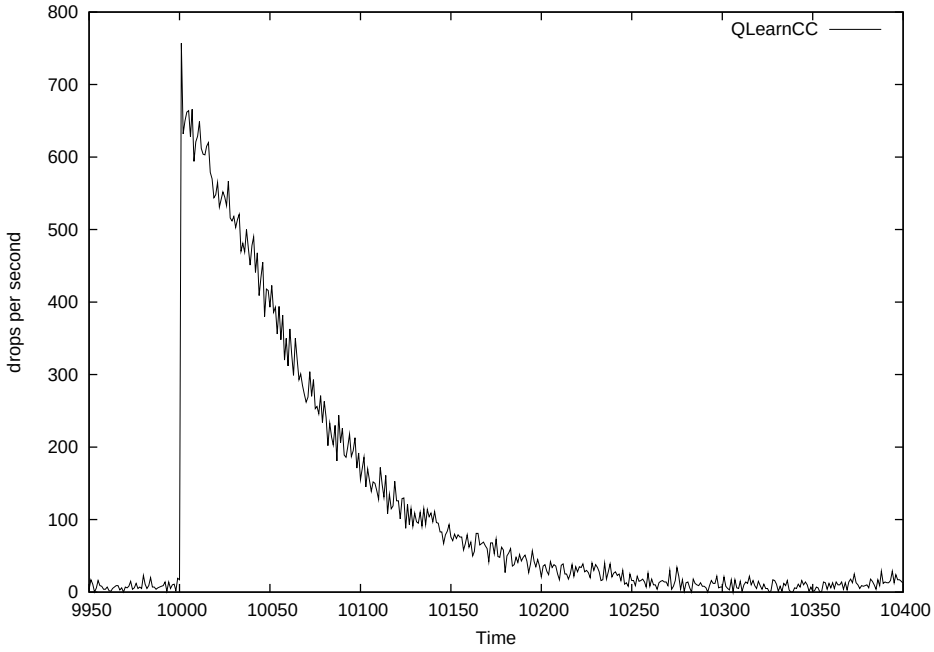


Figure 7.10.: Graph showing the drops per second (packets that would have to be retransmitted) in the overlay using the Q-Learning controller before and after the exchange at the 10000 second mark.

7.4. Tree-based Live Video Streaming

The live video streaming application (Q-Stream) was tested with various configurations. In those configurations where a streaming to each client was possible good trees were constructed and the streaming worked. This was verified by actually watching the resulting streams. It was tested if a streaming server can optimize the trees of a large number of clients and still have enough processing power left to distribute the stream. Given a moderately fast processor it can.
In Figure 7.11 the optimized trees of ten slices are shown. The server bandwidth was set to a relatively low value of 20 MBit/s. The client bandwidths were set to respectively 10 MBit/s. This way the server has to use many clients to distribute the stream.
Clients which disconnect are removed from the server after a maximum time of ten seconds as then the ping timeout occurs. This can of course be configured to be more strict. Q-Stream is able to cope with churn.
The exploration packets provide input to the bandwidth estimation via the Q-Learning congestion controller and allow the Q-Stream server to estimate the latencies between the clients. The experiments confirm that this works. Together with the tree optimization this results in a very efficient streaming technique.

7.4.1. Advantages and Disadvantages

We have shown that the solution to the congestion control problem with multiple, parallel paths can be used to efficiently gather information about nodes. This information can be used to construct trees for live video streaming that, given our perhaps limited knowledge, are as optimal as possible. We can now summarize the advantages and disadvantages of this streaming method:

Disadvantages

- The performance of this streaming approach depends on how good the information we gathered is, e.g. how good the estimated latencies between the nodes are. Only then we can do informed decisions. This becomes increasingly difficult with the number of clients. The more clients we have the more bandwidth has to be inefficiently spent to 'explore' the latencies and bandwidths of the clients, to be able to make an informed decision. An example: If we have 200 clients and want a sample between each to estimate the latencies between them and use 4 hops for the exploration packets. We have $1000 * 999$ different latencies to sample, each exploration packet gets 3 new estimates for the latencies. So we have to at least send about 999000 packets to get *one* estimate for the latencies. With a packet size of 1500 bytes and 10 MBit/s exploration bandwidth we would need 6.5 minutes for that. So clearly there is a limit for how much nodes a streaming server can handle with a given bandwidth – this limit does not exist in other approaches like mesh based streaming or SplitStream as there the clients organize themselves.

- The streaming server has to do more computationally intensive tasks like the tree optimizations mentioned above. This means that we need more powerful hardware depending on the number of clients on the server. In the other approaches this is

done distributed as well. In this approach the streaming server does everything. The clients simply do what the streaming server says they should do and are not able to modify the streaming structure at all – other then by joining or leaving.

- There is currently no tit-for-tat system implemented like in BitTorrent[2] where clients which do not contribute are punished, but clients with a lower amount of shared bandwidth are automatically attached lower in the trees, thus, in a way, punishing them with a higher latency. They should be excluded from the stream in favor of new clients which are willing to forward more streaming data. Clients which cheat by forwarding exploration packets but not the streaming packets can be detected by the negative acknowledgments and should be punished (by exclusion). Currently the congestion controller decreases their forwarding bandwidth and they then eventually become a leaf node (where they belong).

Advantages

- Fast reaction time to quitting clients. Quitting clients get discovered fast. Either they do not react to pings or often they even disconnect themselves via TCP. The trees instantly get reorganized and the streaming can continue without much of an interruption.

- The server can build optimal streaming trees given that he has enough knowledge about his clients. This should be better than the state of the art methods SplitStream or PRIME, given that the number of clients is not that big (see disadvantages above).

- The server can react to situations where there is not enough bandwidth available to stream to all clients. It can send clients to another server or reduce the overall streaming quality.

- Lower delay for the live stream than in mesh based streaming, because the streaming packets get pushed down from the root to the clients in the tree, whereas the mesh based approach works by first announcing which stream packets are wanted and then receiving them. This adds delays of respectively about half of one round trip time for each hop from streaming server to client to the live stream.

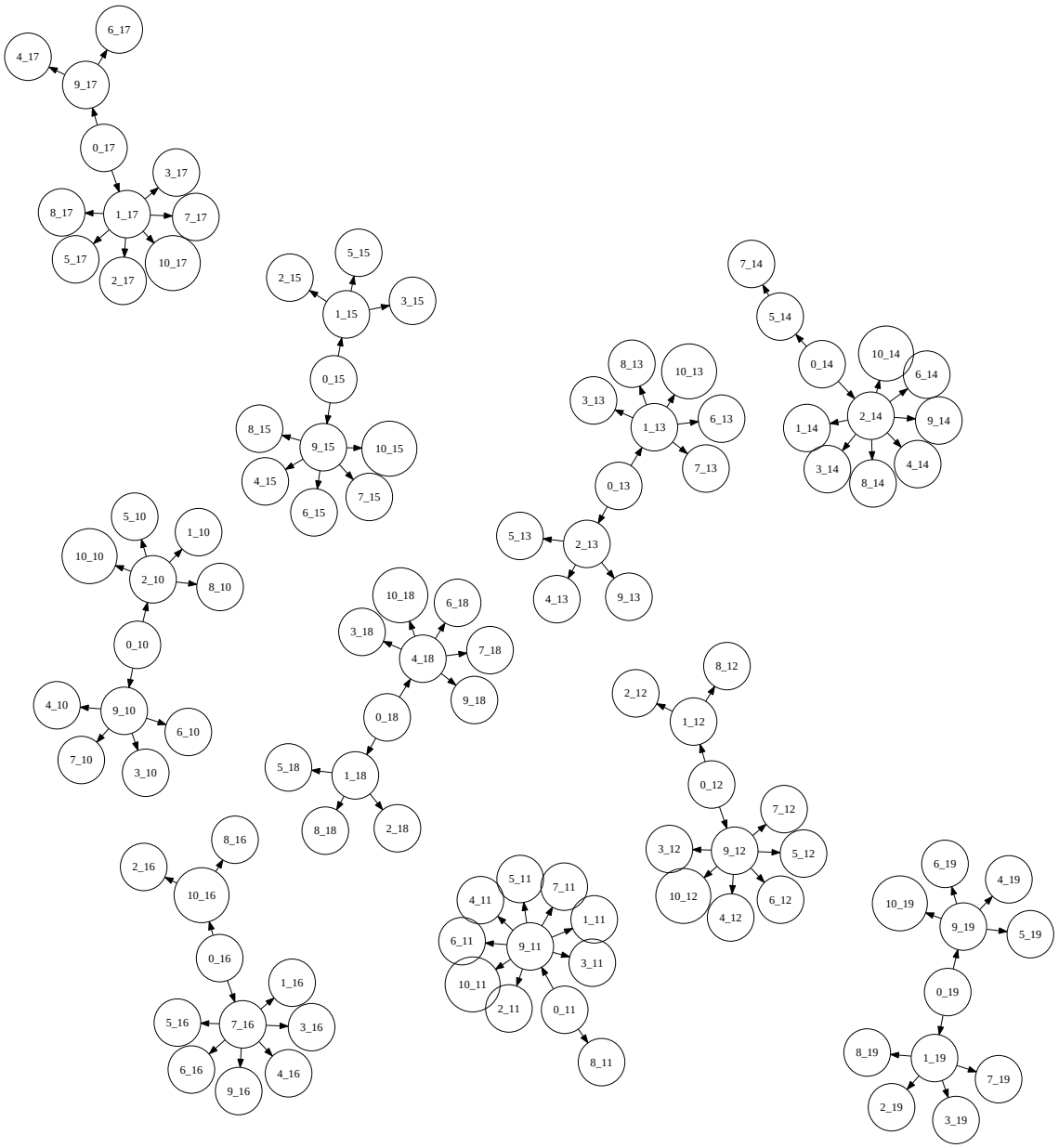


Figure 7.11.: Trees produced by the Q-Stream server for ten slices (slices 10-19). The server was set to a total bandwidth of 20 MBit/s the clients to respectively 10 MBit/s. The latencies between clients are random. The naming of the nodes is as follows: $< nodeid > \_ < slicenumber >$, whereas the node with id 0 is the streaming server itself and all other nodes are the clients.

Chapter 8.
Conclusion

Looking closer at this problem was worthwhile. There are obvious applications (anonymous information sharing and video streaming) where we can use approaches that solve the congestion control with multiple parallel paths problem. We have seen that not using any congestion control gives unusable results – one of the shown approaches should be used: The AIMD congestion controller or the Q-Learning congestion controller. In this chapter it is summarized what these approaches can do. It is shown which of these two approaches is better in certain situations and how the congestion controllers could be further improved.

8.1. Which congestion controller to use?

Which congestion controller to use depends on the specific situation, namely on the number of peers, bandwidths of these peers, delays between peers and desired number of hops. In figures 7.4-7.8 in section 7.3.1 one can see how changing these parameters affects the congestion controllers. Dependencies of these parameters on each other were not considered. Using these plots one can say approximately which situations are affecting the congestion controllers in what way.
The AIMD controller is much worse than the Q-Congestion controllers in situations where there is a lot of bandwidth available at each peer and/or only few peers are participating. With increasing number of peers this advantage gets smaller and one can say it is preferable to use the AIMD controller then because it is more lightweight (uses less memory). It gets worse, however, when the number of hops for each packet is increased or the delays between peers are high. In such cases the Q-Learning controller should be preferred.
An experiment where the bandwidths of the peers change suddenly (section 7.3.3) shows that the Q-Learning approach behaves poorly if it learned about the client bandwidths and then those change. If such change occurs frequently one should use the AIMD approach.
Another experiment shows that the slow start mechanism in the AIMD approach can cause the performance of that approach to be worse if new peers often join the network. This mechanism works better in the Q-Learning approach than in the AIMD approach. It follows that if the number of participating peers is moderate and the overlay is subject to much churn one should use the Q-Learning approach.

8.2. Future work

Other approaches that use Q-Learning as to congestion control like [24] do it using fuzzy logic. Here for the sake of simplicity only plain Q-Learning was used with discrete actions and states. With increasing bandwidth having discrete actions uses a lot of memory. To avoid this, one could have a continuous state and a more advanced reinforcement learning algorithm that uses e.g. neural networks or fuzzy logic to work with such continuous states. The actions then should also be continuous leading to a congestion controller that can react faster to changing network conditions. It was shown that currently the Q-Learning approach reacts badly to such a change in environment. One could also investigate if it is possible to find good signal for such an abrupt change. If it is detected one could reenter the fast start phase to reach better adaptivity to rate changes.
One could investigate if algorithms like Cubic TCP or Compound TCP would improve the performance of the AIMD approach and make it competitive with the Q-Learning approach even with few nodes, long latencies between peers and high bandwidths.
We did not look at how the congestion control algorithms behave in MORE3.1.3 compared to the individual path congestion control that is currently done. This could be an area of further investigation.
The congestion control was done using a fixed number of hops per simulation. One could somehow include the number of hops into the congestion control finding optimal ones. This would also help to be more efficient in situations where the triangle inequality does not hold and more hops are better.

8.3. Summary

We have looked closer at the problem of congestion control with multiple parallel paths. We looked at two applications where an algorithm that solves that problem would be useful.
In anonymous information sharing the application is straightforward: To be anonymous in the Internet one has to send message over multiple hops. To avoid certain attack possibilities where one or more of these hops is the attacker, one has to use multiple paths with multiple hops. Congestion control becomes then necessary to avoid overusing certain nodes.
As a second application video streaming was shown. We claim that video streaming is best done using a client-server system and not using a system of equal peers. Then a method was devised to use clients who are able to relay the streaming data to other clients further increase the number of clients the server is able to serve using the congestion control solution to the multiple path problem.
Following are two possible congestion controllers. Namely the AIMD and Q-Learning approaches. The adaptive increase, multiplicative decrease controller is a simple TCP like approach, which is fast, lightweight and avoids congestion most often. The Q-Learning approach needs more memory and reacts more sluggishly to congestion but learns about which rates each client can handle in a distributed manner and thus reaches more efficient equilibriums than the AIMD approach even with high delays between peers and increasing number of hops. Two non usable congestion controllers were presented: One does nothing at all and just sends the packets to random peers, the other one tries to do perfect congestion control using instant knowledge from all peers (non-distributed solution). The

'perfect' and 'random' congestion controller are used to evaluate the performance of the Q-Learning and AIMD approach.
The parameters of the AIMD and Q-Learning controllers were optimized in many simulation runs. Afterwards they are compared. The result is that the AIMD approach is worse than the Q-Learning approach, if the number of nodes is low and the bandwidths of these hops is high. With increasingly high number of peers or smaller peer bandwidths the AIMD approach becomes more attractive. With an increasing number of hops or with fever peers with higher bandwidths the Q-Learning approach becomes very attractive. In the simulated scenario it performed as well as the perfect congestion controller and thereby doubled the result of the AIMD approach. The results have shown that doing no congestion control is very bad and results in a very high packet loss rate and very few throughput. In an experiment it was looked at how the controllers handle sudden changes in peer bandwidths and there the AIMD controller copes better than the Q-Learning controller, which has a hard time forgetting previously gained knowledge and learning about the new situation.

Bibliography

[1] Sandvine Intelligent Broadband Networks, "2008 Analysis of Traffic Demographics in North-American Broadband Networks," 2008.

[2] B. Cohen, "Incentives build robustness in BitTorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, 2003.

[3] V. Jacobson, "Congestion avoidance and control," *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 1, pp. 157–187, 1995.

[4] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, p. 21, USENIX Association, 2004.

[5] P. Syverson, D. Goldschlag, and M. Reed, "Anonymous Connections and Onion Routing," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, p. 44, IEEE Computer Society, 1997.

[6] O. Landsiedel, A. Pimenidis, K. Wehrle, H. Niedermayer, and G. Carle, "Dynamic Multipath Onion Routing in Anonymous Peer-To-Peer Overlay Networks," in *IEEE Global Communication Conference (GlobeCom), Washington DC*, 2007.

[7] L. De Cicco, S. Mascolo, and V. Palmisano, "An experimental investigation of the congestion control used by Skype VoIP," *Wired/Wireless Internet Communications*, pp. 153–164, 2007.

[8] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," in *IEEE INFOCOM*, vol. 2, pp. 519–528, 2000.

[9] B. Fortz and M. Thorup, "Optimizing OSPF/IS–IS Weights in a Changing World," *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, vol. 20, no. 4, 2002.

[10] H. Wang, H. Xie, L. Qiu, Y. Yang, Y. Zhang, and A. Greenberg, "Cope: Traffic engineering in dynamic networks," in *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, p. 110, 2006.

[11] A. Elwalid, C. Jin, S. Low, and I. Widjaja, "MATE: multipath adaptive traffic engineering," *Computer Networks*, vol. 40, no. 6, pp. 695–709, 2002.

[12] S. Kandula, D. Katabi, B. Davie, and A. Charny, "Walking the tightrope: Responsive yet stable traffic engineering," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, p. 264, 2005.

[13] S. Fischer, N. Kammenhuber, and A. Feldmann, "Replex: dynamic traffic engineering based on wardrop routing policies," in *Proceedings of the 2006 ACM CoNEXT conference*, pp. 1–12, 2006.

[14] V. Raghunathan and P. Kumar, "Wardrop routing in wireless networks," *IEEE Transactions on Mobile Computing*, vol. 8, no. 5, pp. 636–652, 2009.

[15] P. Key, L. Massoulié, and P. Towsley, "Path selection and multipath congestion control," in *IEEE INFOCOM 2007. 26th IEEE International Conference on Computer Communications*, pp. 143–151, 2007.

[16] K. Tan and J. Song, "A compound TCP approach for high-speed and long distance networks," in *Proc. IEEE INFOCOM*, 2006.

[17] H. Han, S. Shakkottai, C. Hollot, R. Srikant, and D. Towsley, "Overlay TCP for multi-path routing and congestion control," *IEEE/ACM Trans. Networking*, 2006.

[18] F. Kelly and T. Voice, "Stability of end-to-end algorithms for joint routing and rate control," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 5–12, 2005.

[19] R. Rejaie, M. Handley, and D. Estrin, "RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet," in *IEEE INFOCOM*, vol. 3, pp. 1337–1345, 1999.

[20] R. Sutton and A. Barto, *Reinforcement learning: An introduction*. The MIT press, 1998.

[21] J. Boyan and M. Littman, "Packet routing in dynamically changing networks: A reinforcement learning approach," *Advances in Neural Information Processing Systems*, pp. 671–671, 1994.

[22] M. Lestas, A. Pitsillides, P. Ioannou, and G. Hadjipollas, "Adaptive congestion protocol: A congestion control protocol with learning capability," *Computer Networks*, vol. 51, no. 13, pp. 3773–3798, 2007.

[23] K. Hwang, M. Hsiao, C. Wu, and S. Tan, "Multi-agent Congestion Control for High-Speed Networks Using Reinforcement Co-learning," *Advances in Neural Networks–ISNN 2005*, pp. 379–384, 2005.

[24] L. Xin, J. Yuanwei, and S. Nan JIANG, "A Genetic Based Fuzzy Q-Learning Flow Controller for High-Speed Networks," *JOURNAL EDITORIAL BOARD*, p. 84, 2009.

[25] X. Li, Y. Jing, G. Dimirovski, and S. Zhang, "Metropolis Criterion Based Q-Learning Flow Control for High-Speed Networks," *The 17th IFAC World Congress (IFAC WC 2008)*, 2008.

[26] N. Magharei and R. Rejaie, "PRIME: peer-to-peer receiver-driven mesh-based streaming," *IEEE/ACM Trans. Netw.*, vol. 17, no. 4, pp. 1052–1065, 2009.

[27] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or multiple-tree: A comparative study of live p2p streaming approaches," in *IEEE INFOCOM 2007. 26th IEEE International Conference on Computer Communications*, pp. 1424–1432, 2007.

[28] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: high-bandwidth multicast in cooperative environments," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, p. 313, 2003.

[29] S. Murdoch and G. Danezis, "Low-cost traffic analysis of Tor," in *2005 IEEE Symposium on Security and Privacy*, pp. 183–195, 2005.

[30] R. Johari and D. Tan, "End-to-end congestion control for the Internet: Delays and stability," *IEEE/ACM Transactions on Networking (TON)*, vol. 9, no. 6, p. 832, 2001.

[31] H. Low, O. Paganini, and J. C. Doyle, "Internet congestion control," *IEEE Control Systems Magazine*, vol. 22, pp. 28–43, 2002.

[32] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[33] B. Hajek and T. van Loon, "Decentralized dynamic control of a multiaccess broadcast channel," *IEEE transactions on automatic control*, vol. 27, no. 3, pp. 559–569, 1982.

[34] F. Kelly, "Stochastic models of computer communication systems," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 47, no. 3, pp. 379–395, 1985.

[35] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on networking*, vol. 1, no. 4, pp. 397–413, 1993.

[36] A. Mondal and A. Kuzmanovic, "Removing exponential backoff from TCP," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 17–28, 2008.

[37] A. Varga, "The OMNeT++ Discrete Event Simulation System," *Proceedings of the European Simulation Multiconference (ESM'2001)*, June 2001.

[38] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," in *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pp. 79–84, May 2007.

List of Figures

List of Tables

Appendix

Algorithm A.0.1 Algorithm for finding a perfect suitable path

Input:

- All nodes N
- Mean round trip times from i to j: $R_{i,j}$
- Current rates for nodes i at time slice t: $R_c^{t,i}$
- Rate limit for nodes i: R_l^i
- Source node $S = A$ and target node $T = B$
- Desired number of hops h. And the current number of hops $h_c = 1$
- The length of the message l
- The current time t
- An empty path P

Output: Path $P = (A, H_0, ..., H_{h-i}, B)$ (indexed by $P_0 = A$ and $P_{h+1} = B$) or $P = \emptyset$ if no path could be found.

```
1:  function GETPATH(h_c, S, T, t, P, l)
2:      if h_c > h then
3:          P_{h_c} = T return P
4:      else
5:          for all n ∈ N do
6:              if n ≠ S ∧ n ≠ P then
7:                  ct = t + (1/2) R_{S,n}
8:                  if R_c^{ct,n} + l ≤ R_l^n then
9:                      d = R_{h_c}
10:                     R_{h_c} = n
11:                     F = GETPATH(h_c + 1, n, T, ct, R, l)
12:                     if F ≠ ∅ then
13:                         P_{h_c − 1} = S return P
14:                     end if
15:                     R_{h_c} = d
16:                 end if
17:             end if
18:         end for
19:     end if
20:     return ∅
21: end function
```

Algorithm A.0.2 Optimized algorithm for finding a perfect suitable path

Input:

- Same input as algorithm A.0.1
- The length of a time slice T_s
- Upper and lower time limits of node n with h hops at time slice t. Intialized as $T_u^{t,h,n} = 0$ (upper) and $T_l^{t,h,n} = \infty$ (lower)

Output: Path $P = (A, H_0, ..., H_{h-i}, B)$ (indexed by $P_0 = A$ and $P_{h+1} = B$) or $P = \emptyset$ if no path could be found.

1: **function** GETPATH(h_c, S, T, t, P, l)
2: **if** $T_l^{t,h_c-1,S} \leq t \leq T_u^{t,h_c-1,S} \wedge |T_u^{t,h_c-1,S} - T_l^{t,h_c-1,S}| < T_s$ **then**
3: **return** $\emptyset$
4: **end if**
5: **if** $h_c > h$ **then**
6: $P_{h_c} = T$ **return** P
7: **else**
8: **for all** $n \in N$ **do**
9: **if** $n \neq S \wedge n \neq P$ **then**
10: $ct = t + \frac{1}{2} R_{S,n}$
11: **if** $R_c^{ct,n} + l \leq R_l^n$ **then**
12: $d = R_{h_c}$
13: $R_{h_c} = n$
14: $F = \text{GETPATH}(h_c + 1, n, T, ct, R, l)$
15: **if** $F \neq \emptyset$ **then**
16: $P_{h_c-1} = S$ **return** P
17: **end if**
18: $R_{h_c} = d$
19: **end if**
20: **end if**
21: **end for**
22: **end if**
23: **if** $R_u^{t,h_c-1,S} < t$ **then**
24: $R_u^{t,h_c-1,S} = t$
25: **end if**
26: **if** $R_l^{t,h_c-1,S} > t$ **then**
27: $R_l^{t,h_c-1,S} = t$
28: **end if**
29: **return** $\emptyset$
30: **end function**

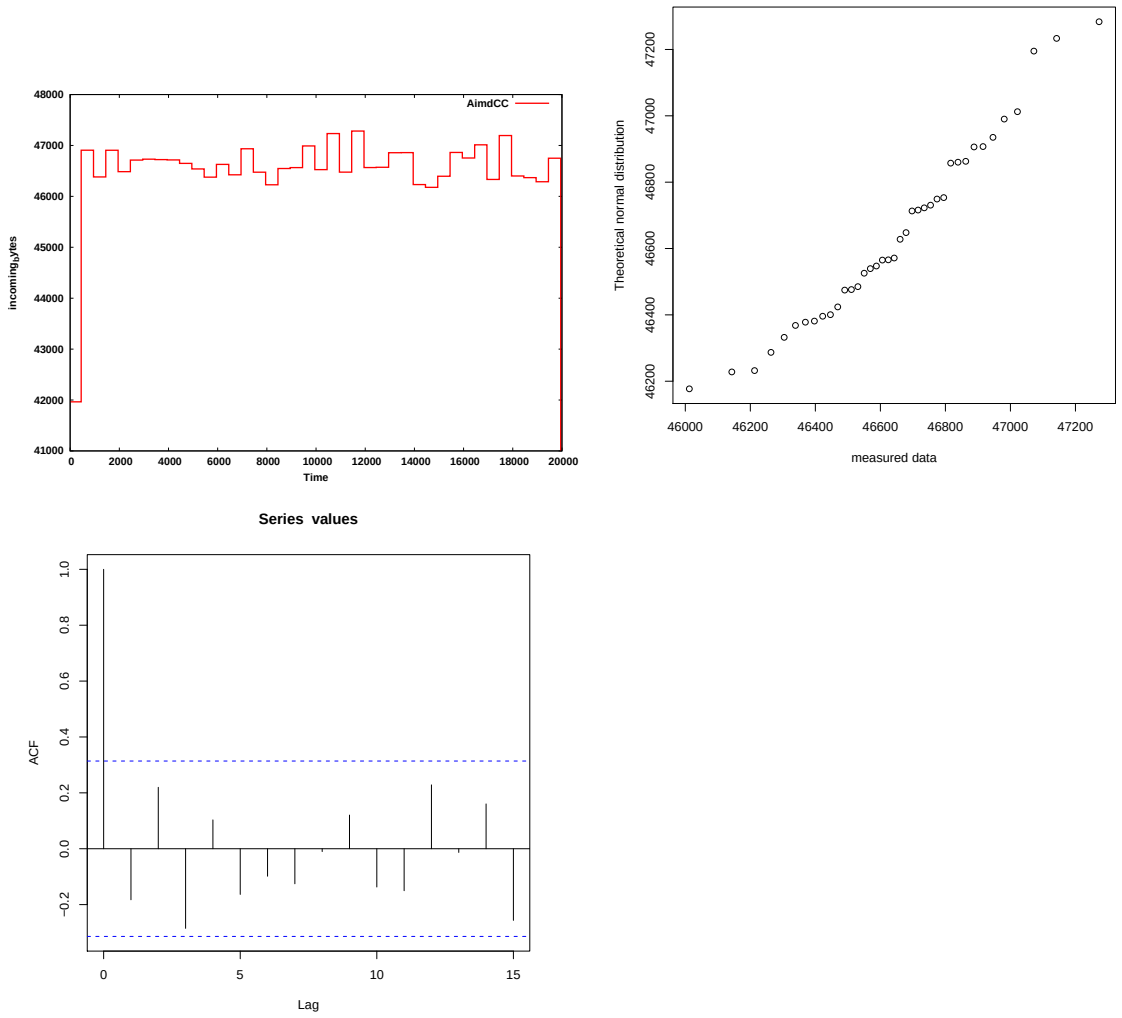


Table A.1.: Simulation of 20 nodes for 200000 seconds. Top left: Arithmetic average of respectively 2000 data points. The means of the non-relay received bytes per second (Data that can be used at tier1 level) are shown. Top right: Quantile-Quantile plot of the data from plot on the left (the first two data points excluded) to the normal distribution. The data is normal distributed. Bottom left: Autocorrelation plot of the data from the top left plot (the first two data points excluded). The data is not autocorrelated.

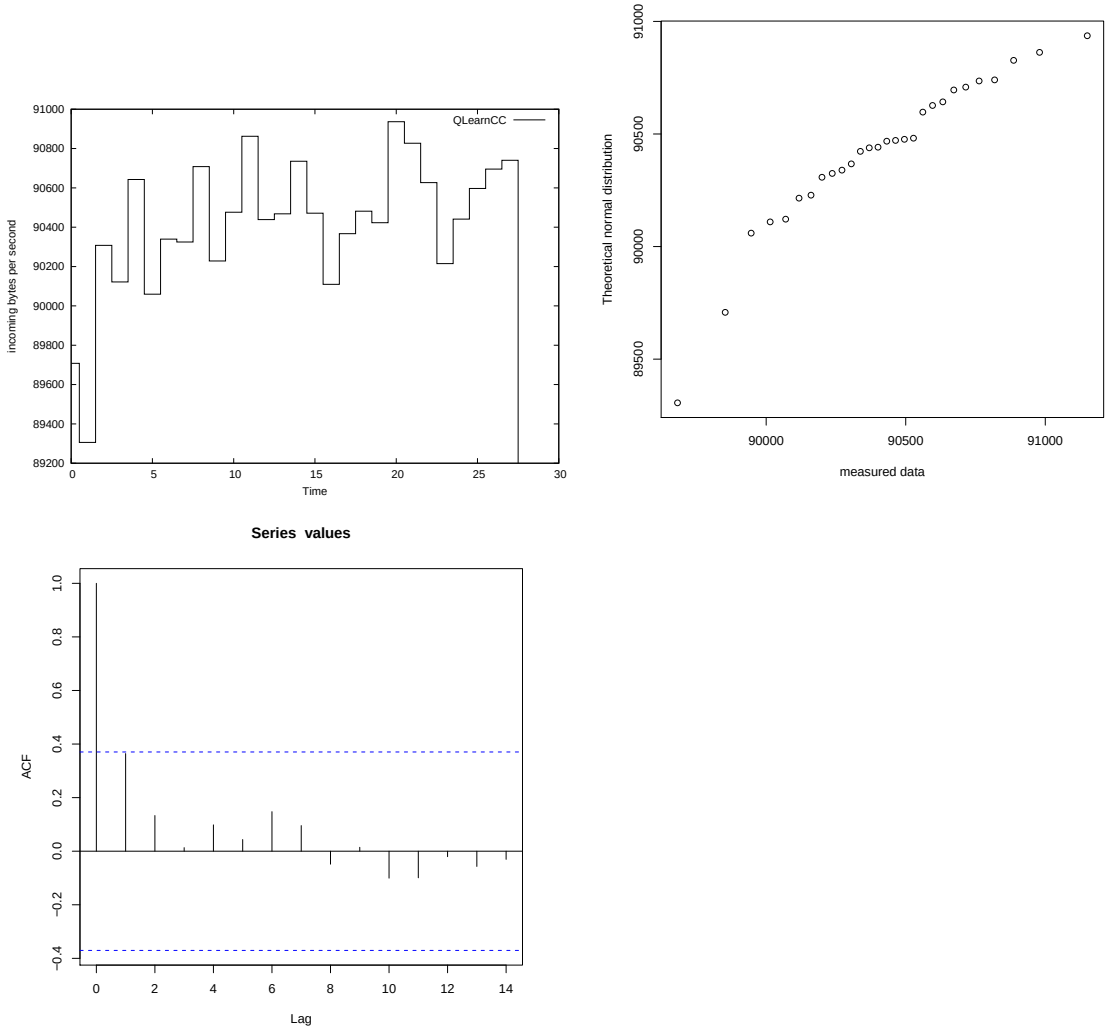


Table A.2.: Simulation of 20 nodes for 300000 seconds. Top left: Arithmetic average of respectively 10000 data points. The means of the non-relay received bytes per second (Data that can be used at tier1 level) are shown. Top right: Quantile-Quantile plot of the data from the plot on the left to the normal distribution. The data is normal distributed. Bottom left: Autocorrelation plot of the data from the top left plot. The data is not autocorrelated.

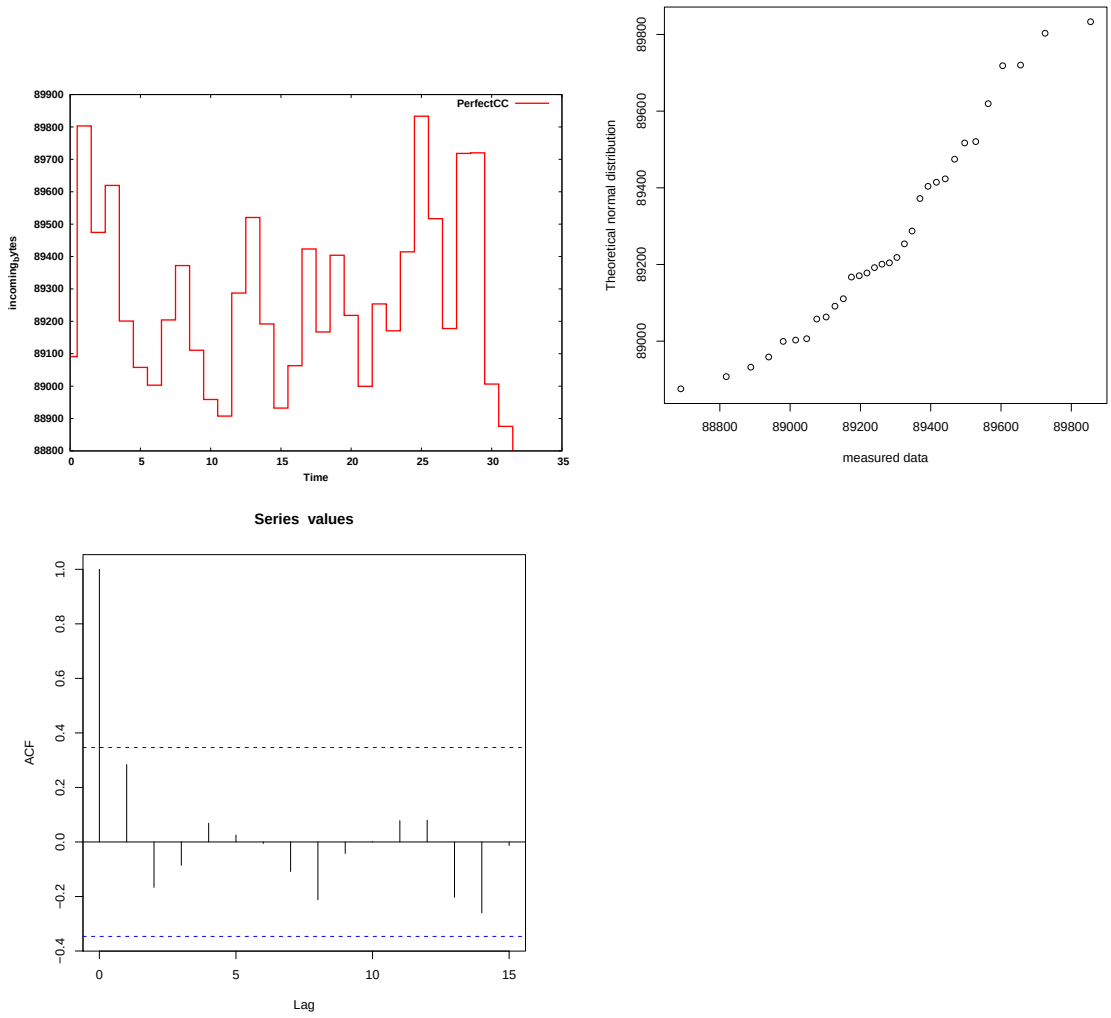


Table A.3.: Simulation of 20 nodes for 1000 seconds. Top left: Arithmetic average of respectively 30 data points. The means of the non-relay received bytes per second (Data that can be used at tier1 level) are shown. Top right: Quantile-Quantile plot of the data from the plot on the left to the normal distribution. The data is normal distributed. Bottom left: Autocorrelation plot of the data from the top left plot. The data is not autocorrelated.

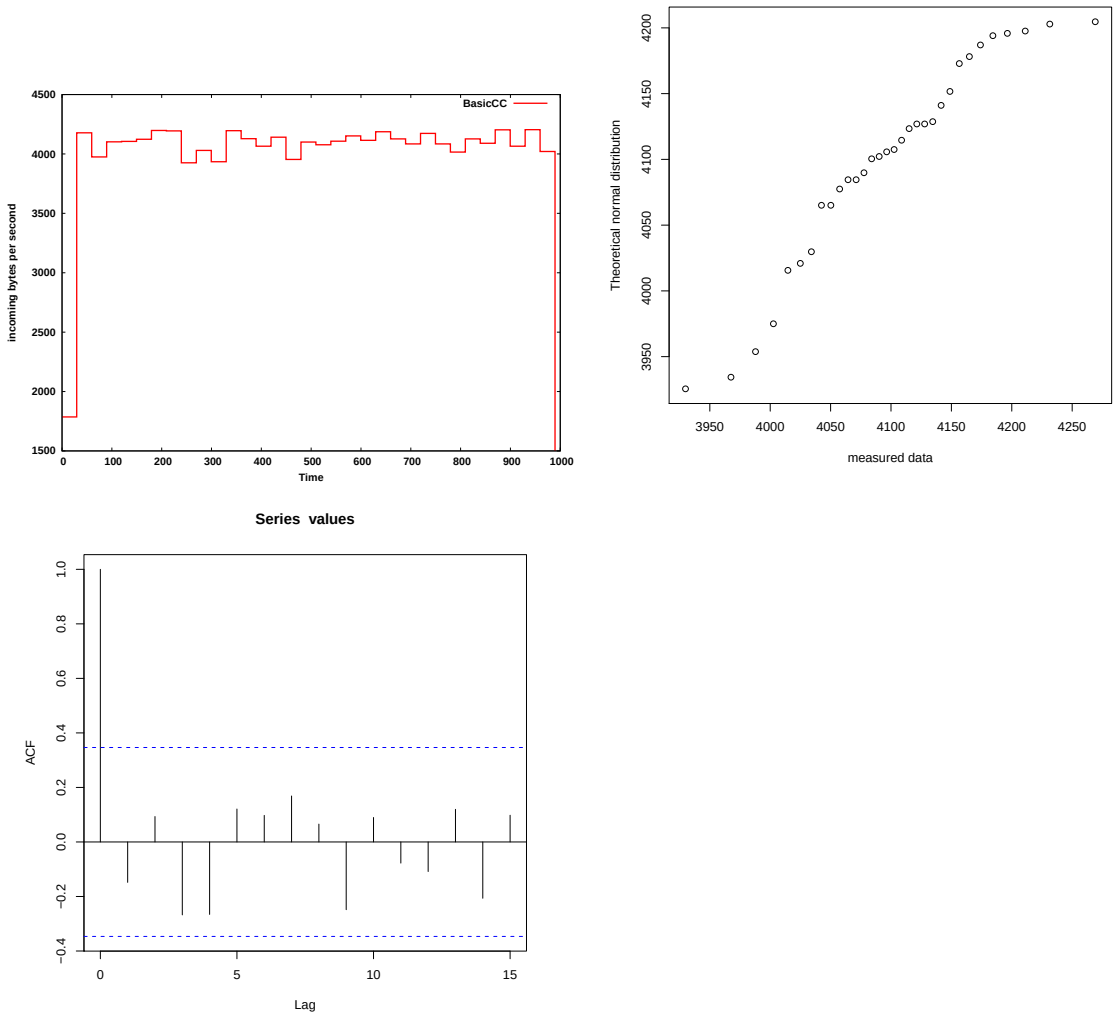


Table A.4.: Simulation of 20 nodes for 1000s. Top left: Arithmetic average of respectively 30 data points. The means of the non-relay received bytes per second (Data that can be used at tier1 level) are shown. Top right: Quantile-Quantile plot of the data from the plot on the left (without first value) to the normal distribution. The data is normal distributed. Bottom left: Autocorrelation plot of the data from the top left plot (without first value). The data is not autocorrelated.

"When I wrote this, only God and I understood what I was doing. Now, God only knows."

Karl Theodor Wilhelm Weierstraß