Path Management for Consistent Reliable Communication in a Multipath mmWave Proxy

Tine Margretha Vister

Thesis submitted for the degree of
Master in Programming and System Architecture
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2022

Path Management for Consistent Reliable Communication in a Multipath mmWave Proxy

Tine Margretha Vister

Abstract

mmWave is a wireless technology that can achieve a speed of multiple gigabits per second, but it is sensitive to blockages. If the signal is blocked, the capacity can drop from Gbit/s to Mbit/s. To mitigate this, multiple connections, i.e. multipath, can be used. A high data rate can be achieved by simultaneously sending traffic over multiple paths. If one path experiences a blockage, a new path can be added to achieve a consistently high data rate.

One way to achieve multipath in a mmWave network is to add a proxy between the sender and the receiver that can provide reliable consistent communication. The proxy will have one connection to the sender and multiple mmWave connections to the receiver. A proxy that supports multi-connectivity has to provide functionality that supports this. One of these functionalities is path management, which is responsible for maintaining the paths in the system.

This thesis will focus on path management. Two new path management schemes are proposed, which are an extension of the two predictive-based path managers proposed in [3]. The new models use a hybrid approach, where both a predictive and reactive control is used. A prediction-based path manager predicts how the queue in the proxy will behave and act accordingly. For example, if it predicts that the queue will increase, a path will be added to prevent the queue from growing. By adding a reactive control, unexpected queue growth can be mitigated by allowing the proxy to add a path sooner, thus draining the queue. To evaluate the performance gains of using the proposed schemes, a comparative simulation study is done first. After this, an emulation study is done.

A proxy is developed to test the different path management schemes in an emulated environment. The proxy contains multiple modules, with path management being one of them. The original schemes from [3] plus the two new models are implemented in the Julia programming language. Multi-threading is an essential part of the proxy, hence a lot of effort is put into making the different modules run concurrently. The proxy is evaluated in a simple testbed.

The evaluations of the path management schemes proposed in this thesis show that they perform better than other path management schemes when tested in a simulated environment. Results from emulation experiments confirm the observations from the simulation study.

Acknowledgments

I would like to thank my supervisors Özgü Alay, David Hayes and David Ros, for their great guidance and support while working on this thesis.

To my family who has always supported and motivated me every day. Your reassuring words has meant a lot to me.

Finally, to my friends at the university. Without you, this process would not have been the same. I am grateful that we have been in this together.

Tine Margretha Vister
May 27, 2022

Contents

List of Figures

List of Tables

Acronyms

4G 4th-generation.

5G 5th-generation.

ACK acknowledgements.

BLEST BLocking ESTimation based scheduler.

BS base stations.

CDF cumulative distribution function.

CWND congestion window.

FCT fast concurrent transfer.

FEC forward error correction.

FIFO First-In First-Out.

FPT first passage time.

GSO Generic segmentation offloading.

HoL Head-of-Line.

IDS Intrusion Detection System.

IETF Internet Engineering Task Force.

IPsec internet protocol security.

LoS Line of Sight.

LTE Long-Term Evolution.

LwPEP Lightweight Performance Enhancing Proxy.

MACT Multipath Based Adaptive Concurrent Transfer.

minRTT minimum RTT.

MKL	math kernel library.
ML	machine learning.
mmPEP	mmWave Performance Enhancement Proxy.
mmTCP	mmWave TCP.
mmWave	millimeter wave.
MPQUIC	Multipath QUIC.
MPTCP	Multipath Transmission Control Protocol.
MTU	maximum transmission unit.
NAT	Network Address Translation.
NIC	network interface card.
NLoS	Non-line of Sight.
NR	New Radio.
OBR	Offloading by Restriction.
PEP	Performance Enhancement Proxies.
PL	path loss.
QoS	Quality of Service.
QUIC	Quick UDP Internet Connection.
RCT	reliable concurrent transfer.
RR	Round-Robin.
RTT	Round-trip time.
SDL	Specification and Description Language.
SNMP	Simple Network Management Protocol.
SPEP	Splitting Performance Enhancement Proxies.
TCP	Transmission Control Protocol.
UDN	Ultra Dense Network.
UE	user equipment.

Chapter 1

Introduction

The world has become more connected since the advancement of the Internet in the 90s, and even more so now that everyone has at least one connected device on them at all times. How they are connected to the Internet may vary, but most probably it is through a WiFi connection or a cellular connection. As more users connect and more applications demand higher data rates, the load on existing wireless technologies increases. Traditionally, wireless technologies are much more limited by the medium they operate in than wired technologies are, resulting in that they cannot achieve as high data rates as a wired connection can. To achieve a bandwidth of several gigabits, a cabled connection that can offer a consistently high speed had to be used. A new wireless technology called millimeter wave (mmWave) mitigates this by providing gigabit data transfer with low delays [1]. This is made possible by the short wavelengths and high frequencies. Applications that aim at running on mobile devices (e.g. a smartphone) that need high bandwidth to function properly will benefit significantly from mmWave capacity. This can, for instance, be augmented/virtual reality media, emergency communications, or 360-degree streaming video, i.e. applications that require high data rates with low delay.

Operating at mmWave frequencies does bring some challenges that need to be addressed before becoming a useful technology. First, mmWave suffers from path loss (PL) to a much higher degree than other wireless communications [1, 2]. This can be mitigated by using beamforming technologies. Another problem with mmWave links is their sensitivity to blockages. If a person or a wall comes between the end-user and the antenna, the signal strength will decrease dramatically. This will lead to wide fluctuations in the capacity because when there is no blockage, the data rate can be gigabit high, but when there is a blockage, the data rate can drop to megabit or lower. This is very challenging for communication protocols, especially the ones that need reliable consistent communication, i.e. steady transmission rate with low delay. A solution to the blockage problem can be to use multiple mmWave paths to reduce the possibility of complete unavailability. If one of the paths experiences some blockage, the other paths are likely to be in Line of Sight (LoS).

Two ways of implementing a multipath solution exist: *Above-the-Core* and *Core-Centric* [2]. In Above-the-Core integration, the multipath is implemented at each end system without impacting the network. In Core-Centric integration, the connection between the server and the client is split, with a proxy deployed in the network being between them. A single path connection is used between the server and the proxy, and multiple connections are used between the proxy and the client. The Core-Centric approach is more likely to be adopted in 5G systems since it enables more control of the multi-connectivity.

Figure 1.1 depicts a mmWave scenario where the multipath solution uses a Core-Centric integration to implement multi-connectivity. The proxy is placed at the edge of the Internet cloud, where the user connects to the internet via five mmWave connections to the proxy. The proxy consists of multiple components that deal with packets' transmission and the management of the paths.

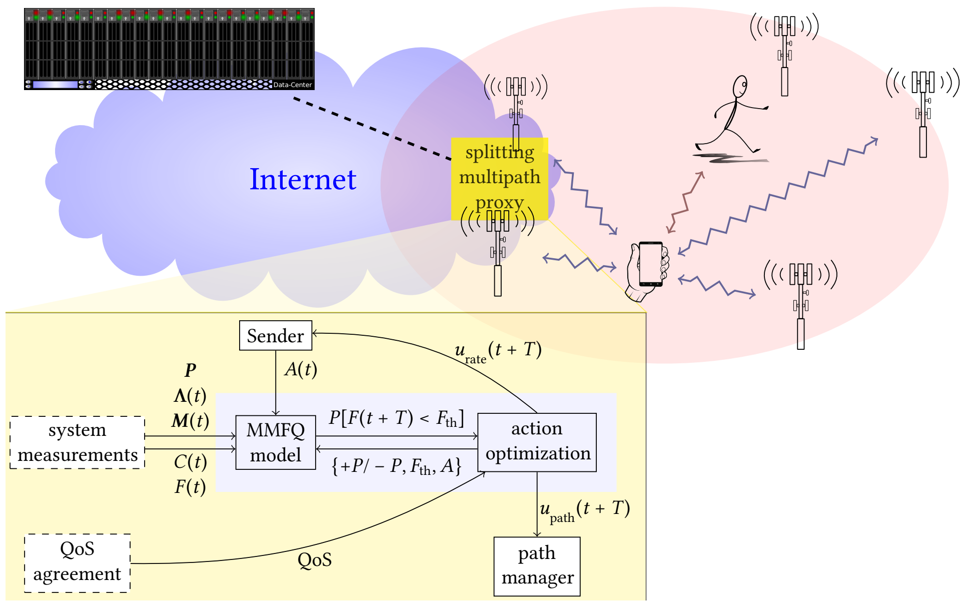


Figure 1.1: System architecture overview of mmWave scenario with five base stations connected to the multipath proxy. Figure taken from [3].

A multipath proxy deployed in a 5G network that splits the connection between the sender and the receiver has to work with minimal overhead, i.e. the Quality of Service (QoS) should not degrade but rather be improved. To do this, the proxy has to deal with the potential of wide fluctuations in the capacity due to the intermittent availability of the links to the client. A way of dealing with this is to let the proxy store packets coming from the server, and then sending them when the links towards the client are available again. However, there is a tradeoff between storing too many packets in the proxy and having low delay. The proxy must sustain the high capacity while keeping the delay at a minimum.

Multipath protocols typically have four key functionalities: path management, packet scheduling, congestion control, and reliable transfer [2]. The path manager is responsible for managing the paths, the packet scheduler is responsible for sending packets over the different paths, the congestion control is responsible for detecting congestion in the network and adjusting the sending rate accordingly, and finally, the reliable transfer is in charge of loss detection and loss recovery. All of these have to be optimized in a mmWave proxy for it to be as little overhead in the proxy as possible.

Out of the four core functionalities that a multipath proxy has to provide, path management will be the focus of this thesis. One way of reducing the need for a large buffer in the proxy is to manage the different paths in a smart way. A path manager that can add and remove paths dynamically as the available capacity of the different mmWave links changes will yield higher performance in terms of higher bandwidth since it can add a new path if a path is in Non-line of Sight (NLoS), i.e. the capacity decreases dramatically since there is a blockage between the base station and the user equipment (UE).

[3] proposes two new path management models that are prediction-based. Since a mmWave network can achieve a sending rate of several Gbit/s if the paths are in LoS and down to Mbit/s or even kbit/s when the paths are in NLoS, the queue will quickly fill up if a server is sending traffic with a speed of Gbit/s when the paths are in NLoS. The idea is to predict how the queue will look in the near future and, based on this, add/remove a path. If it predicts that the queue will grow, it can add a path before the queue starts to fill up, and hopefully, the queue will not grow as much.

This thesis proposes two new path management models that are based on the models proposed in [3]. They are prediction-based but have a backup plan in case the predictions of the queue do not correspond with reality. The predictions of the queue happen periodically. But, if the queue suddenly fills up between these intervals, a reactive control can be triggered that will add a path immediately to prevent the queue from filling up.

The path managers proposed in this thesis will be tested in a simulator written by the authors of [3]. After this, they will be tested in an emulated environment. For them to be tested in an emulated environment, a proxy has to be implemented. A way of implementing such a proxy is presented in this thesis. The proxy aims to operate in a mmWave network, where the data rates are high. It has to deal with wide fluctuations in the available capacity toward the receiver. For the proxy to be able to handle this, running things in parallel is required, hence multi-threading is an essential part of the proxy. Furthermore, how to optimize the proxy so that it can handle the high data rates going in and out is discussed.

The proxy's goal is to provide reliable consistent communication with high bandwidth and low latency to applications. It is important to note that the applications of interest are not necessarily TCP-based, hence, the proxy has not been designed as a TCP-centric performance-enhancing proxy. The proxy will run on a testbed that was created in collaboration with one of the supervisors of this thesis, David Hayes. A lot of effort was put into making the testbed as similar to a mmWave network as possible. This includes changing the capacity of the connections, to simulate a moving person in an urban landscape where the LoS/NLoS characteristics of a link change as the user moves around.

1.1 Problem statement

A mmWave connection can reach a capacity of multiple Gbit/s when it is in LoS, but only Mbit/s or lower when it is in NLoS. This will impact the queue in a multipath proxy to a very high degree if not dealt with properly. One way of dealing with this is to manage the different subflows smartly to reduce the possibility of a growing queue.

1.2 Research Questions

The goal of this thesis is to answer the following research questions:

RQ1: How can a path manager with both a reactive and a predictive control be designed?

RQ2: How can a proxy be implemented to work in a mmWave network?

RQ3: Is the Julia programming language an adequate choice for implementing a real-world proxy?

RQ4: How do different path managers perform in a proxy that operates in an emulated environment?

1.3 Structure

This thesis is constructed as follows:

Chapter 2 – Background This chapter gives an introduction to the most relevant technologies that this thesis will use, as well as a walk-through of related work to get a better understanding of the research field that this thesis is related to.

Chapter 3 – Path manager models In this chapter, five different path manager models are presented, where two of them are new models presented in this thesis. The models presented here are the ones that will be tested later.

Chapter 4 – Simulation-based performance evaluations This chapter includes a comparative simulation study, where the different path manager models are tested in a simulated environment. The models will be compared to get a first overview of how the models proposed in this thesis perform differently from other models. Parts of this chapter have been submitted for publication in section 5.4 of [4].

Chapter 5 – Design and Implementation of the Proxy This chapter goes through the design of the proxy that will be placed in a more realistic scenario, an emulated environment. First, the proxy design will be explained, then how it was implemented with regard to the design.

Chapter 6 – Test environements In this chapter, the virtual and the real testbed are described. A thorough explanation of how the real testbed is built to emulate a mmWave network will be provided.

Chapter 7 – Testing the proxy implementation on an emulated mmWave network This chapter goes through the results of testing the proxy in the testbed. The proxy's performance and how the different path managers affect the performance of the proxy will be included.

Chapter 8 – Conclusion This chapter wraps up this thesis by stating the major findings from the results. The research questions will be answered, and a discussion of what future work can consist of is provided.

Chapter 2

Background

In this section, the three main technologies that are relevant to this project will be presented. First of all, this thesis will look into wireless communication using the new up-and-coming mmWave technology. Secondly, the proxy developed in this project will be deployed between the wireless mmWave medium and the wired internet. The most prominent type of proxy used in such a scenario is Performance Enhancement Proxies (PEP), thus PEP will be presented. Lastly, multiple mmWave paths will be used between the proxy and the UE, hence multipath is a crucial technology for this master thesis.

2.1 Relevant technologies

This thesis will implement a multipath proxy in a mmWave network, hence mmWave, proxies and multipath are relevant technologies that will be discussed in this section.

2.1.1 mmWave

5th-generation (5G) cellular networks are being designed to have low latency and high capacity. 5G offers much higher bandwidths than earlier cellular networks (for example 4th-generation (4G)), thus achieving a much higher data rate than ever before. mmWave is seen as the key enabler for 5G [5]. It is a term used for electromagnetic waves with a wavelength between 1 mm and 10 mm and with frequencies above 28 GHz [1]. With such high frequencies, mmWave can offer gigabit data throughput since it can take advantage of rich spectrum resources [6, 7]. In addition, since mmWave communications operate above 28 GHz, and other wireless communications (for example 4G) operate below 6 GHz, applications that earlier had to share an already congested medium with others [8], now, with mmWave communications, can get a large portion of the medium all for themselves.

Even though the advantage of mmWave is that it can offer such a high throughput because of its high frequency, the high frequency does come with a disadvantage [1, 7]. Radio waves that have high frequency suffer from PL to a much higher degree than radio waves that have lower frequencies. This is because the signal loses its strength quickly when traveling through the air. Beamforming techniques can compensate for this by steering the signal toward the receiver. Still, the beamforming technology is not yet able to realize its potential, but it is expected that it will once the technology matures [9].

In addition to this, since mmWave cannot penetrate through objects like humans or walls, LoS is required to have a connection that reaches the full potential of the mmWave signal. It can function in an NLoS environment if the surroundings are highly reflective, but at the expense of a significant decrease in received power and thus available capacity [8]. The wide fluctuations in received power from LoS to NLoS are also a challenging aspect of mmWave communications. This is one of the critical problems the transport layer has to deal with. One way to cope with the characteristics of mmWave links is to use multiple channels [1]. [7] investigates how tuning different Transmission Control Protocol (TCP) parameters affects the throughput and delay. They found out that implementing a larger buffer will increase the throughput but at the cost of higher latency.

2.1.2 Proxies

Most wireless networks today consist of many different types of middleboxes that are deployed in the middle of an end-to-end connection and perform functions on the packets that are being transmitted [10]. Examples of these middleboxes can be Network Address Translation (NAT) or proxies like firewall and Intrusion Detection System (IDS). A proxy is an intermediate server between the end-user and the remote server. One type of proxy that is mainly used for enhancing wireless links is PEP. A network that performs poorly due to characteristics the network paths have can get improved performance when using a PEP [11]. Such characteristics can be high error rates, links with low bandwidth, or links with asymmetric bandwidth.

There are many types of PEPs, for instance, Splitting Performance Enhancement Proxies (SPEP). SPEP are proxies that split a TCP connection between the mobile terminal and the remote host, ending in one TCP connection from the mobile terminal to the SPEP, and one from the SPEP to the remote host [5]. This allows the creation of two optimized connections for two very different links by tuning appropriate TCP parameters and performing data caching and local retransmissions. This can be especially useful for mmWave communication, where the connection is split between a wired and wireless part. The two types have very different characteristics, for instance, availability. There exist several novel TCP SPEPs for mmWave links [12–14] (which will be discussed in subsection 2.2.1), but it is found that current 4G Long-Term Evolution (LTE) SPEPs do significantly increase the performance of a transmission, if the transfer is small [5]. This will be of great importance when transitioning to 5G, where a heterogeneous network will include both 4G and 5G traffic.

An argument against using PEPs is that they break the end-to-end semantics of a connection [11]. This means that certain required end-to-end functions cannot be correctly performed since only the end system can execute them. An example of such a function is the internet protocol security (IPsec), which provides secure communication by encrypting the packets. By breaking the end-to-end semantics, the use of end-to-end IPsec is disabled because only the end-systems can encrypt and decrypt the packets. The packets have to be visible for the PEP so that it can do its job. When the PEP cannot examine the header of the packets being transmitted, since IPsec is hiding the content, it will function poorly or not at all. One downside to PEP is therefore that IPsec cannot be used since the end system has to trust the proxies in the network to use it, which it generally cannot do. To have a secure connection while using a PEP, it must be implemented at a higher level, e.g. the application layer. This will mean that the transport layer headers will be exposed, but the data would be secured if measures are taken at the application layer.

Another issue with PEP is scalability [11]. Since PEPs operates above the IP layer, it requires more processing power than a router. In addition, since PEPs requires a per-connection state, it needs more memory than a router. All of this leads to that a PEP can not have as many connections, thus reducing its scalability. A solution to this can be to add several PEPs, where the combination of them results in a higher number of possible connections. This will, however, increase the system's complexity, making it less feasible.

2.1.3 Multipath

A client or a server connected to several interfaces can gain higher throughput and/or reliability by using several of the interfaces simultaneously. To increase the throughput, a client that is connected to several interfaces (e.g. mmWave and 4G) can get higher performance by sending and receiving traffic over all the available interfaces. To increase reliability, the host can send all traffic over the main channel, and the other channels can be used to send corrections packets. Correction packets can either be a copy of already sent packets or a coded packet that can be used to recover the lost packets. It is also possible to dynamically change from one method to the other (from increased throughput to increased reliability and vice versa), as done in [15].

Another benefit multipath has is that the host gets increased resilience towards failure in the network. This is because the availability increases when there are several paths that traffic can be sent over. If a link on one path fails, the traffic can be redirected to other operational paths.

Four different components are needed when implementing a multipath protocol: a path manager, a packet scheduler, congestion control, and reliable transfer [2]. The path manager's job is to keep track of all the paths, including their availability and metrics like the available capacity. The packet scheduler then uses this information to decide which path the traffic should be sent over. The congestion control is responsible for detecting congestion in the network and adjusting the sending rate. The reliable transfer is responsible for detecting any losses and recovering them.

The most dominant multipath protocol over the transport layer today is the Multipath Transmission Control Protocol (MPTCP). MPTCP was standardized in 2013 and in 2020 an updated version was published by the Internet Engineering Task Force (IETF) [16]. MPTCP operates over TCP, where it creates a new TCP connection for every available interface.

The goal of MPTCP is threefold [2, 17]. 1. Improve the throughput, meaning that it should perform at least as well as a single path TCP. 2. It should not use more resources than the standard TCP under similar conditions. 3. Steer packets toward less congested paths. It has been proven that MPTCP performs better than original TCP when using paths with similar characteristics, but it fails to outperform it in heterogeneous networks [18].

Another multipath protocol that is gaining popularity is Multipath QUIC (MPQUIC), which utilizes multiple Quick UDP Internet Connection (QUIC) paths. QUIC is an emerging transport layer protocol that offers encrypted, multiplexed low-latency data transfer [19]. In contrast to MPTCP, MPQUIC does not require changes to the operating system, making it easy to deploy. In addition, it does not have the three-way handshake TCP has, but a faster setup phase. This will decrease the time it takes to set up new paths. Lastly, MPQUIC can differentiate the different flows so that if a packet gets lost in one of the paths, the other paths will be unaffected. This will eliminate the Head-of-Line (HoL) blocking problem that MPTCP suffers from. All considered, MPQUIC gives higher performance compared to MPTCP [19].

2.2 Related Work

This section will go through related work, to get a better understanding of the research field that this thesis is related to.

2.2.1 Transport Layer Proxies

Several published papers focus on designing proxies used for high-bandwidth and low latency networks, such as 5G. In [12], the authors present mmWave Performance Enhancement Proxy (mmPEP), which is a novel TCP design for mmWave communications. The proxy is placed between the wired and wireless network and sends early-acknowledgements (ACK) to the server, thus breaking the end-to-end TCP semantics of the connection. Furthermore, it performs batch retransmission, meaning that it sends the lost packets, but it also sends a certain number of following packets since they are expected to be lost. mmPEP tries to take advantage of the short period between LoS and NLoS, hoping that LoS will come soon and not slow down the TCP sender. In the evaluation of mmPEP, they only considered data rates up to 100 Mbps, which is normally obtained during NLoS. In addition, they did not look at latency, only the throughput and the delivery ratio, hence not considering the bufferbloat problem (high latency because of excessive use of buffers).

Another proxy is proposed in [13], which is called milliProxy. milliProxy aims to fully utilize mmWave communications' capacity while being transparent to the end-users and respecting the end-to-end semantics of a TCP connection. It includes three different modules: flow buffer, flow window management, and ACK management. The flow buffer is used to store the packet's payload before it can be forwarded to the client. The flow window management module controls the amount of data sent to the receiver. Finally, the ACK management module is used to inspect incoming ACKs and clear the corresponding packets in the buffer so that new packets can be transmitted. In the evaluation of milliProxy, one can see that it successfully reduces the sending rate of the TCP receiver, thereby overcoming the bufferbloat issue. Moreover, the results show that milliProxy can faster reach full utilization after an NLoS period, resulting in higher goodput. On the other hand, milliProxy focuses on a single-UE case, but the proxy should likely be able to serve multiple UE's simultaneously.

The novel TCP framework from paper [14] called mmWave TCP (mmTCP) focuses on achieving the full potential of mmWave communications by forwarding packets from the server to multiple UE's as fast as possible. To do this, two functions are used: batch retransmissions and online cache management. Batch retransmission works the same way here as it did in mmPEP, i.e. sends multiple packets at once if a packet is lost. The online cache management algorithm is a cache that adaptively adjusts depending on the current cache and the channel status of each UE. This means that if one UE is in an NLoS period, mmTCP can automatically reallocate some of the cache allocated to this UE to another UE that is in LoS. Another mechanism the cache has is called *cache reservation* and ensures that new UE's that connect to mmTCP get a small portion of the cache. The simulation on mmTCP shows that it improves the end-to-end TCP throughput in various scenarios, both in single-user cases and in multiple-user cases, compared to conventional TCP and cache-enabled TCP (the same as mmTCP, but without batch retransmissions).

All of the above papers presented novel protocols that work over TCP. It is a well-known problem that TCP is hard to extend, and there exist many harmful middleboxes that intercept the transmission and can obtain the unencrypted TCP header [20]. Google's QUIC is a new transport protocol aimed at overcoming this issue by encrypting both the payload and the header. This means that middleboxes, like PEP, will become useless as is. The authors of [21] therefore present their prototype implementation of their Lightweight Performance Enhancing Proxy (LwPEP), which aims at supporting both TCP and QUIC traffic. It checks incoming packets and determines if it is TCP traffic or QUIC traffic. Based on the type of traffic, different actions are taken. If it is TCP traffic, the LwPEP can forward packets to the client and ACKs to the server. If it is QUIC traffic the LwPEP does not have access to the packets nor the ACKs. They propose that instead of sending sequence numbers in the form of ACKs back to the server, the proxy can instead send parts of the original server message back to the server. The server can then use this sliced payload and look it up in a table where a mapping from the sliced payload to the sequence number is stored. The author's measurements show that LwPEP adds a 0.06 ms delay to the end-to-end connection for TCP and as much as 0.2 ms for QUIC. The delay can be further decreased by implementation in hardware.

All of the proxies mentioned above are built on TCP (one supports QUIC as well), however, the proxy implemented in this thesis is not tailored for TCP traffic. Furthermore, as the proxy aims at providing reliable consistent communication, keeping the delay at a minimum is essential. To do this, the buffer in the proxy can not be too big, as this will increase the delay. Finally, the proxies mentioned use only one path from the proxy to the receiver, thus, they only need to consider flow control. However, the proxy developed in this thesis considers multiple paths toward the UE. This means that in addition to implementing a flow control mechanism, also path management has to be included. In the next section, several multipath protocols are presented to explain the possibilities and challenges of multipath.

2.2.2 Multipath protocols

As mentioned earlier, multipath can increase the throughput and reliability. Several papers have introduced multipath protocols especially designed for 5G communications. In paper [15], they introduce Multipath Based Adaptive Concurrent Transfer (MACT), which is designed for transmission of real-time video streaming, which is a similar application that the proxy developed in this thesis aims at serving. MACT tries to maximize the transmission rate and the reliability by dynamically switching between fast concurrent transfer (FCT) and reliable concurrent transfer (RCT). In FCT the packets are sent over all the available paths, while in RCT the packets are sent over one path, and copies of the packets are sent over the other paths. However, the authors did not use 5G links in the evaluation of MACT. They tested it using two other wireless technologies: WiFi and cellular.

In [22] the authors looked at the possibility of using multipath where both 5G New Radio (NR) links and LTE links are used. The scheme they propose is a new MPTCP offloading scheme, called Offloading by Restriction (OBR). The goal of OBR is to offload data from the NR path to the LTE path when the NR path experiences a long NLoS period. This scheme is not suitable for applications that require an overall high data rate since the transition from using a NR path to a LTE path will lead to a significant drop in bandwidth, however taking action when a path experiences an NLoS period is similar to what the proxy proposed in this thesis does.

The proxy developed in this thesis will operate in a mmWave network, thus the challenges of using multiple wireless technologies at once will not be addressed. Furthermore, the proxy aims at dynamically changing the number of operational paths based on their capacity. This will be possible if there is a dense deployment of mmWave base stations (BS), ultimately creating a Ultra Dense Network (UDN). The user will then likely be connected to one or more BSs, and as the user moves around, new BS will appear as others are blocked. In [23] the authors investigate several possible backhaul schemes that can handle the large data flow coming from the UE going to the internet. This is out of scope for this thesis but is something that needs to be addressed before deploying a mmWave-based UDN.

2.2.3 Multipath management

When using multiple paths simultaneously, paths must be selected and established by a path manager. The path manager decides when and how new additional paths will be established, as well as how paths are being torn down [3]. There are many different path managers, and which one to select generally depends on the application's requirements. MPQUIC has a path manager where both hosts can negotiate the multipath capabilities. This enables hosts to tell the state of a path or claim a preference for a path. For example, a host can communicate the state of a path to another host, i.e. tell if the path is available, on standby or abandoned.

The other multipath protocol that is widely used, MPTCP, uses another approach. It has three different path management implementations available: *default*, *ndiffports* and *full-mesh* [2, 3, 24]. *default* does nothing else than to accept the creation of new subflows passively. *ndiffports* uses the same IP-address pair for all of its paths, but different TCP ports. This will enable the connection to be perceived as several TCP connections instead of only one, thus preventing bandwidth-limiting middlebox interference. Finally, *full-mesh* is a path manager that establishes all possible sub-flows. This is especially useful in Internet scenarios where applications that want to increase the reliability or throughput choose to use all available paths.

In [25] they present MPTCP-MA, which aims at improving the performance of MPTCP during intermittent path connectivity. MPTCP-MA estimates the status of a subflow by using MAC-layer information. This can be used to manage the different subflows of an MPTCP connection and direct traffic over the subflows with higher signal strength, thereby increasing the throughput. The concept of managing the paths based on their signal strength is similar to the way the proxy in this thesis manages its paths based on LoS/NLoS characteristics, however, it is not based on TCP traffic.

Path management can, in addition to deciding how to create and tear down paths, also include choosing the best available path [3]. Handover management is one type of algorithm that fall under this domain. When only one path is active, and the other paths are on standby, a handover manager is needed to change the active path. This handover can be between a 5G and 4G link or between only 5G links.

2.2.4 Packet scheduling in multipath protocols

Packet scheduling is a central task in a multipath protocol. The scheduler's job is to forward packets over the different available paths. This section presents several multipath schedulers that can be applied to any multipath protocol. The most basic algorithm is Round-Robin (RR). It cyclically sends packets over each path, but only if there is space in the congestion window (CWND). The CWND indicates how much the sender can send before receiving an ACK. RR do not consider the characteristics of each path before sending a packet over. This means that if one path is in an NLoS period, this algorithm will still try to send packets over that path until the CWND is adjusted. This can lead to many packet drops, which will yield low performance.

Another path scheduler scheme is minimum RTT (minRTT). minRTT takes the Round-trip time (RTT) characteristics into account, along with the CWND availability, when choosing a path for a packet. First, it sends packets over the path with the lowest RTT until the CWND is full, then it sends packets over the path with the second-lowest RTT [26]. This scheme takes better into consideration the heterogeneity of the paths, compared to RR, by sending packets over the fastest path, minRTT reduces the out-of-order packet delivery at the receiver. This will reduce HoL blocking, thus achieving higher throughput.

In [26] the authors proposed a new scheduler called BLocking ESTimation based scheduler (BLEST). It works almost the same way as minRTT, but BLEST tries to reduce the HoL blocking problem even further by estimating if sending more packets over a path will cause HoL blocking. It determines if it is best to send packets on the path with larger RTT or wait to send packets on the path with lower RTT until it becomes available again. In the evaluation of BLEST, they compared it with minRTT among others, and the result was that BLEST outperformed all the other schedulers with better application goodput and lower packet delay.

The schedulers presented above are all non-learning based schedulers. They have a predefined policy they run. However, learning-based schedulers are promising to use in a challenging environment, such as 5G. Peekaboo [27] is a learning-based scheduler. It uses the same logic as the two previous schemes, that is, choosing the path with the lowest RTT. The difference from before is how to choose another path if the path with the lowest RTT has a full CWND. Peekaboo applies a machine learning (ML) algorithm to take this decision, which learns at runtime the best decision to make. This approach works well when the paths are heterogeneous, i.e. the RTT differs from path to path. However, in 5G communication, the RTT over the different paths typically has the same average RTT [28].

In [28] the authors present a modified version of Peekaboo: M-Peekaboo, that combat the issue of similar average RTT over the different paths by selecting a path based on several metrics, like throughput, loss rate, and RTT. The evaluation done in [28] shows that M-Peekaboo performs better than the non-learning based algorithms RR, minRTT and BLEST. One reason is that a scheduler using a learning-based approach can learn the dynamics of the 5G network over time, making it better at making good scheduling decisions.

Chapter 3

Path manager models

A proxy placed between the internet and the end-user needs several features to function correctly (e.g. path manager and packet scheduler). The scope of this thesis is on path management. In this section, five different path managers are presented:

- Reactive control
- CDF-based prediction
- FPT-based prediction
- CDF-based prediction with reactive control
- FPT-based prediction with reactive control

The first three models come from previous work, and the last two are new models proposed in this thesis.

This section will answer the first research question: *How can a path manager with both a reactive and a predictive control be designed?*.

3.1 Reacting to queue changes

The main job of a path manager is to decide how many paths that should be operative. This includes adding and removing paths as needed. For instance, if the queue starts to fill up in the proxy, adding a path can help drain the queue to keep the delay at a minimum. One way of implementing a path manager is to take a **reactive** approach. If the queue starts filling up, the path manager reacts to this and will add a path. This leads to very accurate decisions, but only if changing the number of paths is a faster process than there are significant changes in the network. If the network condition changes more rapidly than the path manager can change the number of paths, it can lead to low performance.

Figure 3.1 shows a simple scenario where the sending rate is higher than the capacity of one path and how the queue behaves when a reactive control is used. Whenever the queue surpasses the high

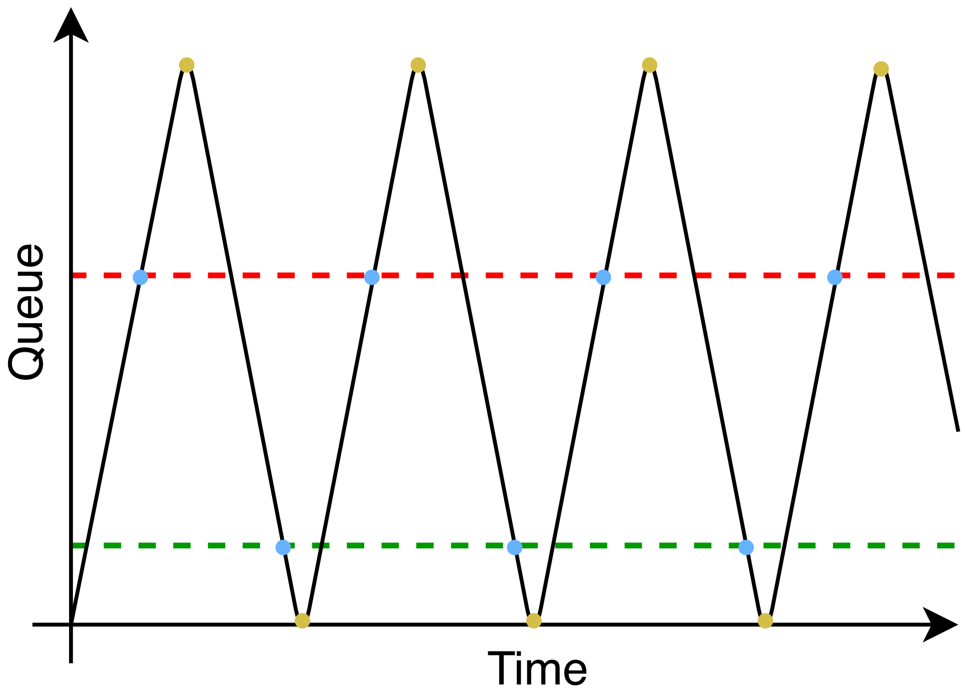


Figure 3.1: Queue dynamics when using a reactive model. This is meant only as an illustration to show how a reactive control model can affect the queue.
The red line is the high queue threshold (add a path if the queue is more than this), and the green line is the low queue threshold (remove a path if the queue is less than this). The blue dots indicate when a reactive control is triggered and the yellow dots is when a path change is performed.

threshold, the red line, then a reactive control will be triggered, marked as a blue dot in the figure. A path gets added by sending a signal to the operator, telling it to add a path, then the operator has to make a path operational. This process takes some time, therefore a path change, the yellow dot, happens some time after the decision to add a path is made, the blue dot. This gives the queue a chance to grow even more. When a path is added, the queue will drain. When the low queue threshold is reached, a path will be removed. Whenever a path is removed, the queue will increase again. The spikes in the queue will be higher as the sending rate increases since more packets will be transmitted. Since mmWave is a network that can reach gigabit capacity, using a reactive control is too slow, since the queue will fill up quickly when it gets a chance to do so.

Moreover, it is not possible to change the number of paths too often, as it is costly for the operator to set up and tear down paths for multiple users simultaneously. Another path management scheme must be used to meet the high data rates that a mmWave network gives.

3.2 Prediction based models

A predictive approach can be used to mitigate the challenges a reactive model has in a mmWave network. This has been done in [3], where the authors present two new path management models. Both are prediction-based, which means that they do not react to changes in the network but rather predict how the network will behave and act accordingly. By trying to foresee the future, instead of responding to the past, a prediction-based path manager can prepare for changes rather than always being a step behind by reacting to changes. This is more suitable for a high data rate mmWave network where changes can happen very suddenly and impact the proxy.

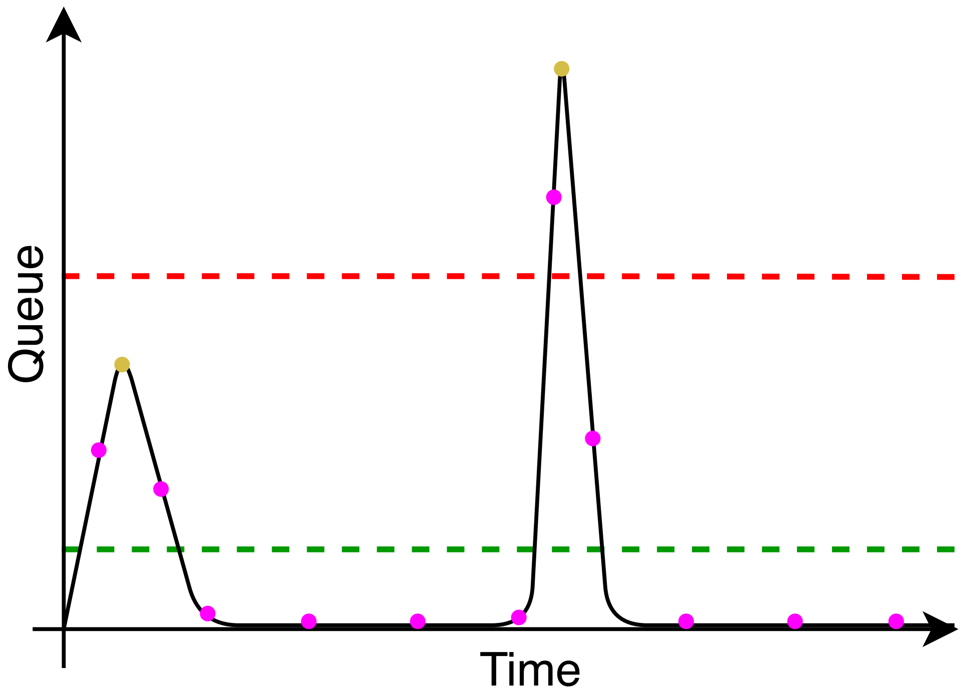


Figure 3.2: Queue dynamics when using a predictive control model. This is meant only as an illustration to show how a predictive control model can affect the queue.
The red line is the high queue threshold (add a path if the queue is more than this), and the green line is the low queue threshold (remove a path if the queue is less than this). The pinks dots indicate when a predictive control is triggered and the yellow dots is when a path change is performed.

Figure 3.2 shows how a predictive-based model may affect the queue. The pink dots indicate that a predictive control is triggered, and is done periodically. When the first predictive control is triggered, the predictions will show that using one path will with high certainty make the queue surpass the high threshold, the red line, thus a path is added to mitigate this. For the next predictive controls, no action is taken, since a path less will make the queue grow, and a path more is not necessary. However, an unexpected queue growth can happen, and since the predictive control is triggered independently from the queue occupancy, a predictive control may not be triggered before the queue reaches, and surpasses, the high threshold. A path will eventually be added so that the queue can be drained.

The authors of [3] suggest that the predictive control is triggered every 150 ms. Whenever a predictive control is triggered, how the queue occupancy will be for the next time horizon, set to 200 ms, is predicted. The predictions will go toward a steady-state queue distribution as one increases the time horizon. This is not useful for control purposes, thus keeping the time horizon not too long is essential.

The LoS/NLoS characteristics of the paths affect the proxy's capacity to send packets to the receiver. Predicting the availability of the different paths, i.e. when a path will be in LoS, will therefore significantly impact the performance of the proxy. If one can predict that a path will be in NLoS, then a path can be added to compensate for the capacity drop when a path gets in NLoS. [3] uses a two-state Markov model for each path to do this. This type of model has been used to model human movement and, in particular, mmWave blocking. Each path will have a 2-state LoS/NLoS continuous-time Markov model, which models the transitions between LoS and NLoS for the given path. When knowing which paths that will be in LoS, and for how long, one can model how the queue will behave in the future. This can predict future queue distributions and find the probability that the queue reaches a certain threshold.

Two different Markov models are used in [3]: the full state Markov model (see Figure 3.3) and a Weighted Queue model with starting state, WQx (see Figure 3.4). The full state Markov model models the LoS/NLoS dynamics for all the operational paths. In Figure 3.3 three paths are operational, where all the different variations of LoS/NLoS dynamics are modeled. λ_i represents the rate of path i moving from NLoS to LoS, and μ_i represents the rate of path i moving from LoS to NLoS. The number of states increases exponentially as the number of paths increases. This also means solving this model will increase rapidly as the number of paths increases, thus it is only used when there are only a few paths.

For control purposes, it is essential that it does not take too long to solve the Markov model. Solving the full Markov model is too time-consuming if the number of paths is high. WQx is used when the number of paths is higher than two. The state of the different paths greatly influences the system's short-term dynamics, thus, knowing the starting state in the full model and a few "hops" from the starting state will be enough to calculate the LoS/NLoS dynamics for the next time horizon. This is done in the WQx model. Figure 3.4 depicts this by having a starting state which determine which path, K, to start at.

Two prediction-based models were presented in [3]. The first one is a distribution-based predictive control: **CDF-based prediction**. This model looks at the probability that the proxy queue distribution, found by using the cumulative distribution function (CDF), is more/less than a particular threshold. This means that an action will be taken

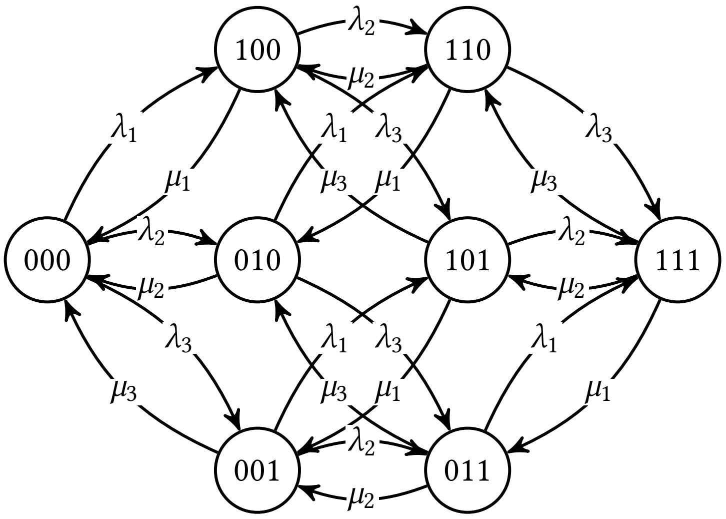


Figure 3.3: Full state Markov model of LoS/NLoS state when there is three operational paths. Each node represent if the path is in LoS - 1, or in NLoS - 0. λ_i is the rate path i change from NLoS to LoS, and μ_i from LoS to NLoS. Figure taken from [3].

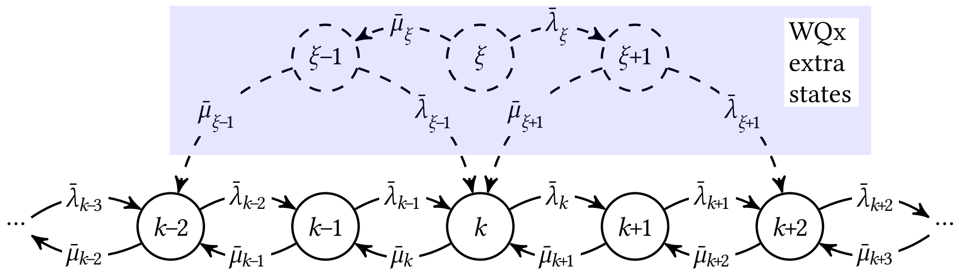


Figure 3.4: Weighted Queue model with starting state, WQx, which has an additional starting state. Figure taken from [3].

if the queue distribution is over/under a threshold. The other path management model that is presented in the paper is one based on first passage time (FPT): **FPT-based prediction**. If the queue crosses a particular threshold within the time horizon, an action will be taken. Both models are probabilistic, which means that some level of randomness is involved when finding a decision to take. For example, the model based on the cumulative distribution function (CDF) will only add a path if there is a more than 1% chance that the queue is over 500 packets and only remove a path if there is more than 99% chance that the queue is under 250 packets. Neither of the models presented was optimized, as the paper's primary goal was to illustrate the feasibility and potential benefits of a predictive control method over a reactive control method.

When to use each of the predictive control mechanisms depends on the QoS requirements the application has. If the application requires a stable network connection but does not mind some delay occasionally, using the CDF predictive control is a good choice. If the application requires only very low delay and consistently high bandwidth, using the first passage time (FPT) is the best choice. This is because it reacts faster than the CDF predictive control since the queue only has to reach the threshold once for something to happen, while the CDF predictive control mechanism has to look at the queue distribution for the whole time horizon.

3.3 Prediction based models with a backup plan

A prediction can never be 100% accurate, and it is bound to be wrong sometimes. A model that solely uses predictions to make decisions can lead to bad decisions. mmWave is a type of network that can quickly fill up the buffer; thus, a path manager taking a wrong decision can quickly increase the delay since the queue can grow at a fast pace. Therefore, two new path managers are presented:

- **CDF-based prediction with reactive control**
- **FPT-based prediction with reactive control**

The first one is based on the CDF-based predictive control model, and the second one is based on the FPT-based predictive control model from [3]. The new models are very similar to the old ones. They use the same Markov model to make the predictions, and they have a control interval every 150 ms where the decision is made based on either CDF or FPT.

The difference from the models presented in [3] is that instead of only relying on the predictions, an additional control instance is added to compensate for unexpected events that will increase the queue and lead to higher delay. If something unexpected happens between the control intervals, the new models will react to this and change the number of paths. They do not need to wait until the next control interval, which makes the path manager more robust to sudden changes in the network. This is illustrated in Figure 3.5, where the same scenario as Figure 3.2 is shown, but because a reactive control can be triggered in between the predictive controls, a path will be added sooner when the queue starts to grow quickly which will prevent the spike from becoming as high.

Figure 3.6 is a Specification and Description Language (SDL) [29] diagram of a predictive control model with a reactive control. In Predict, either CDF- or FPT-based predictive control can be used. Adding a reactive control to a predictive-based model adds some complexity. A path change can be triggered in two different ways in the new models. The main way of changing the number of paths is by a predictive control. When a predictive control is triggered, a decision will be found by executing the prediction. The decision can be to add a path, remove a path or do nothing. Another way a path change can be initiated

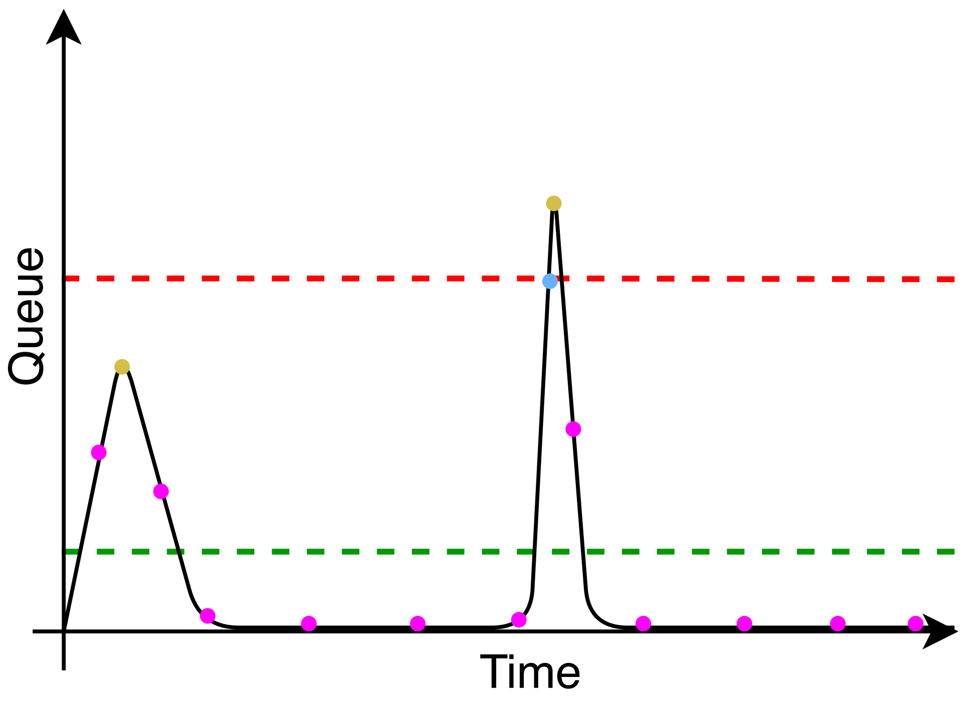


Figure 3.5: Queue dynamics when using a predictive control model with a reactive control. This is meant only as an illustration to show how a predictive control model with a reactive control can affect the queue. The red line is the high queue threshold (add a path if the queue is more than this), and the green line is the low queue threshold (remove a path if the queue is less than this). The pink dots indicate when a predictive control is triggered and the blue dots indicate when a reactive control is triggered. The yellow dots are when a path change is performed.

is by reactive control. Reactive control is only triggered in the most critical scenario when there are insufficient resources and adding a path is necessary. When a reactive control is triggered, a check is done to ensure that there is another path to add. If so, the path with the highest capacity gets added. If there is no new path to add, the reactive control that was triggered will only be ignored.

The predictive control is periodically performed every 150 ms. And after any path change, a new predictive control is scheduled. When a reactive control is triggered, and a path gets added, a new predictive control will be scheduled, effectively shifting all following predictive controls some milliseconds forward. This ensures that a predictive control is not triggered too soon after a path change following a reactive control.

Alg. 1 is the pseudo-code for the new models. Several states are introduced to keep track of when different things are done. First, *t_IC* and *t_PC* are used to know when the last control was and when the next predictive control will be triggered, respectively. *Q_HT* is the higher queue threshold, so a reactive control will be triggered to add a path if the queue is over this. The reactive control interval, T_R, is a time limit used to prevent a reactive control from being triggered too soon after

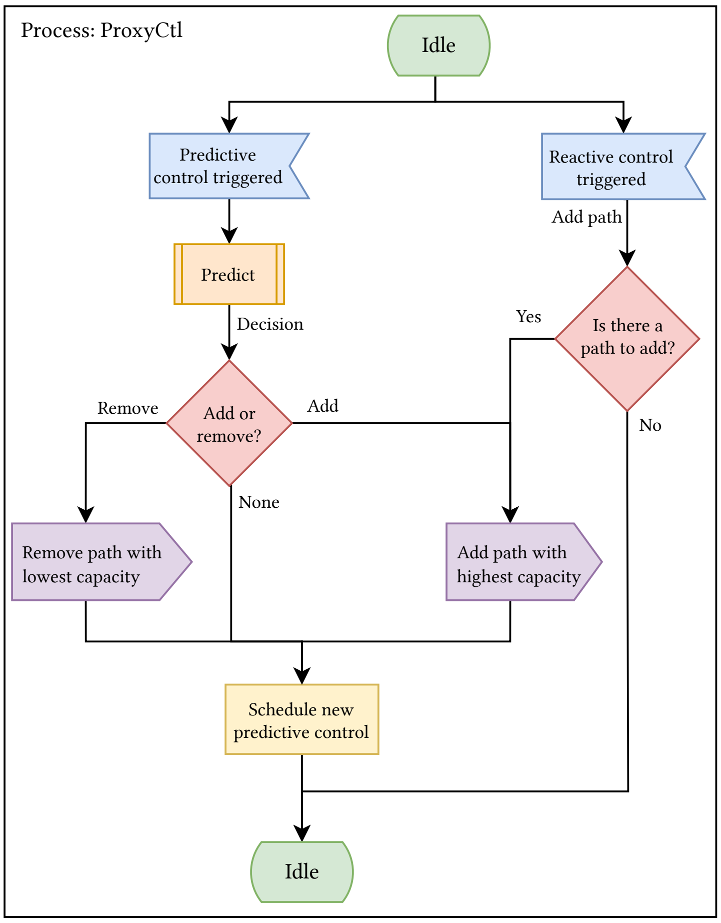

Process: ProxyCtl
Idle
Predictive control triggered
Reactive control triggered
Add path
Predict
Decision
Yes
Is there a path to add?
Remove
Add or remove?
Add
No
None
Remove path with lowest capacity
Add path with highest capacity
Schedule new predictive control
Idle

Figure 3.6: SDL of a predictive based control system with reactive control

another path change. Changing the number of paths too often is costly for the operator; thus, there must be a limit to prevent changing the number of paths too often. *PC_min_T* is the predictive interval. This is set to 150 ms so that a predictive control is triggered every 150 ms.

Control() is the function that will be called every time a predictive or reactive control is triggered so that a path change can be initiated. Predictive control is triggered every 150 ms, thus Control() will be called every 150 ms. Additionally, Control() will be called every time a reactive control is triggered. Reactive control is triggered if both checks are true: 1. ensure that it has gone enough time since the last control. 2. the queue size must be larger than the high queue threshold *Q_HT*. If both are true, then a reactive control will be triggered and *t_IC* will be set to now and Control() will be called.

To differentiate if a predictive or a reactive control has been triggered in the Control() function, *t_IC* and *t_PC* are used. Additionally, since a reactive control will schedule a new predictive control, a way of knowing which predictive controls are still valid is needed. Control() handled the three scenarios in the following way:

Algorithm 1: Predictive control with reactive control

```
1  Every Arriving packet
2      if $now > t\_IC + T_R$ then
           // only do reactive control if we want to increase the
              number of paths
3          if $QueueSize > Q\_HT$ then
4              $t\_IC = now$
5              Control()
6  Every Control
       // reactive control was triggered
7      if $now == t\_IC$ then
8          decision = add path
       // trigger a predictive control
9      else if $now == t\_PC$ then
10         decision = Predict()
       // ignore a predictive control
11     else
12         return;
13     if $decision == add or remove path$ then
14         Send a signal to change paths
15     $t\_IC = now$
16     $t\_PC = now + PC\_min\_T$
17     Schedule Control() in $t\_PC$ time
```

1. If now equals to *t_IC*, a reactive control was triggered, and the decision will be to add a path.

2. If now equals to *t_PC*, a predictive control is triggered. In Predict(), either CDF- or FPT-based prediction can be used. The decision of this prediction can be one out of three: add a path, remove a path or do nothing.

3. If now equal to neither *t_IC* nor *t_PC*, a predictive control was supposed to be triggered but will instead be ignored. This occurs when a reactive control has been triggered, consequently scheduling a new predictive control so that the former predictive control scheduled is ignored when it is activated.

After a decision is found, and it is to either add or remove a path, a signal will be sent to the operator so that the number of paths can change. *t_IC* and *t_PC* will be updated, and a new predictive control will be scheduled in *t_PC* time.

Chapter 4

Simulation-based performance evaluations

The first step in developing a new path manager scheme is implementing it and testing it in a simulated environment. A simulated environment gives maximum control without any interference from unknown sources (e.g. congestion in the network). The chosen simulator for this first study is an event-based simulator. The five different models presented in chapter 3 are implemented in the simulator and tested. Using a simulator to test the models enables an easy way to check for bugs in the code and see if the models work as intended. Furthermore, the different models can be compared to each other. However, the results from this comparison can not be used to conclude that one model is better than another confidently. To do this, the models have to be tested in a more realistic scenario, which is done in chapter 7.

4.1 Implementing the models in the simulator

To simulate a 5G mmWave network, an event-based simulator (also called a discrete-event simulator) is used. An event-based simulator operates by first placing events in a queue and then executing these events chronologically [30]. An event is a change in the simulation environment. For example, sending or receiving a packet is an event. This results in a highly accurate simulation environment. The speed of the execution depends on the complexity of the design and the level of activity within the simulation. Using an event-based simulator can be slow for simulations with many events. However, it brings the simplicity and the repeatability required for this study and the possibility of testing new models.

The new models presented in this thesis are a continuation of previous work done in [3]. The models they presented were tested in a simulator they had written in Julia. The same simulator will be used to test the two new models proposed in this thesis. As the same simulator is used, the reactive model, the CDF-based prediction model, and the FPT-based prediction model were already implemented.

When implementing a model into a simulator, some action must be taken to ensure that the simulation is as realistic as possible. For example, the time it takes to perform actions in a real-world system must be added as a cost in a simulator. This can be the time it takes for the predictive control to do the predictions or the time it takes to change the number of paths.

The two new models presented in this thesis – a predictive control model with reactive control – were implemented in the simulator based on Alg. 1. However, since it is a simulator, some changes had to be made to make the simulations as realistic as possible. One of these things is adding calculation time for the predictive models. Every time Predict() is called, a time penalty must be added to simulate a real-world proxy that needs a little time to run the calculations and get a result.

4.2 Simulation characteristics

Five different models are tested in the simulator:

- Reactive control
- CDF-based prediction
- FPT-based prediction
- CDF-based prediction with reactive control
- FPT-based prediction with reactive control

The three first is written by the authors of [3], while the last two are the new models proposed in this thesis. The simulation setup when testing the models is the same as was used in [3]. This means that the simulation will try to simulate a mobile scenario in an urban landscape, where a user moves around.

The goal is to achieve a steady data rate of 2 Gbps through the system while keeping the delay at a minimum. Throughout the simulation, the channel capacity of each path will vary to simulate the shadow fading effect, i.e. that the capacity fluctuates as objects block and the signal strength vary, as experienced in a real system. In addition, the paths will go from LoS to NLoS and vice versa; this to simulate a moving person moving in a city.

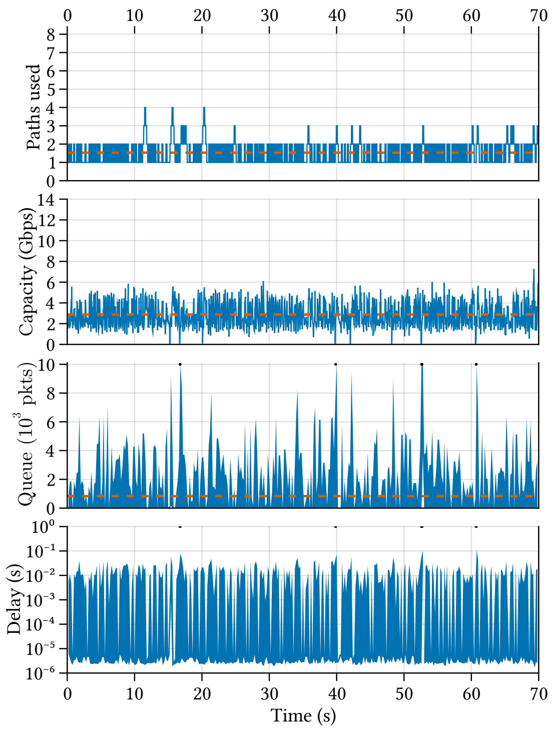
4.3 Results

Different things are tested out in the simulator to get a good overview of how the two new models perform compared to the others. First, the simulator is run once for each of the five models. The results of each of these runs are then compared. Next, the models are tested by running the simulations multiple times, in total 100 times, to get a more accurate comparison between the different models. Finally, the reactive control interval, i.e. how soon after a previous path change a new path change can be triggered by a reactive control, will be explored to see how it impacts the results.

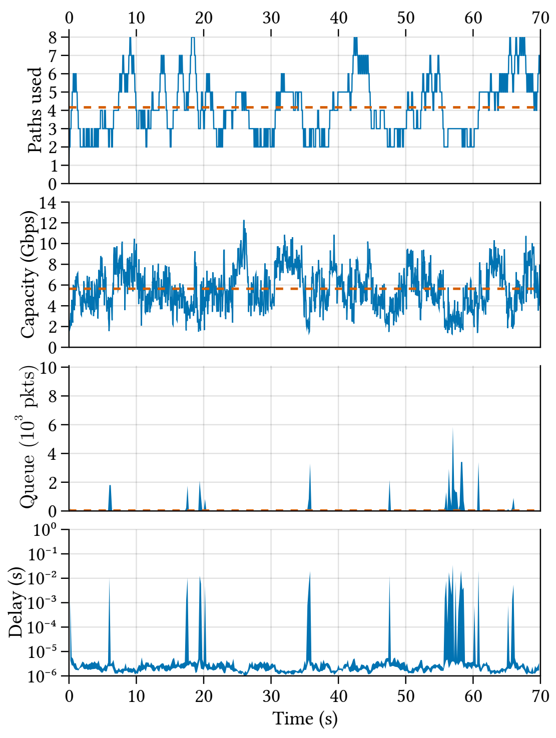
4.3.1 First impressions of the new models

Each model is run through the simulator once, to see if any improvements by using one of the new models can be seen. The reactive control interval, T_R, is set to 25 ms, so that it must go 25 ms before a new path change, initiated by a reactive control, can be initiated after the last path change. This will prevent the reactive control from being triggered too soon, which may be costly for the operator. Figure 4.1 shows the result after one simulation run.

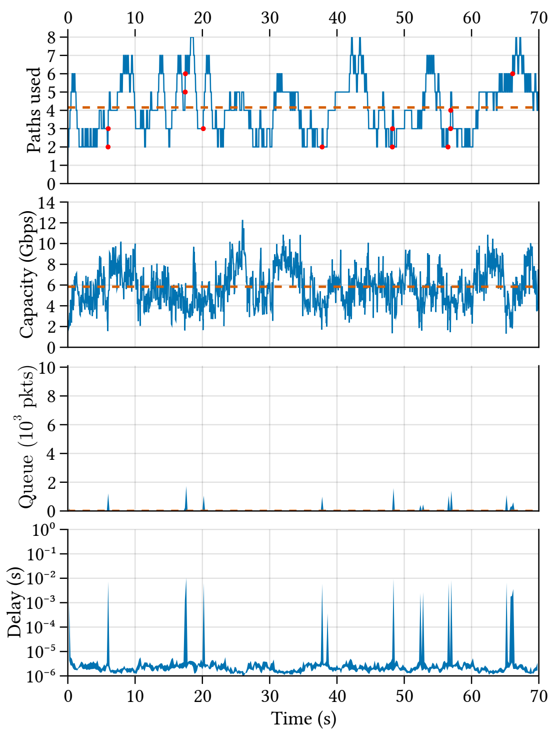
For each model, four different plots are created: the number of paths used, the available capacity based on which paths are in use, the queue size, and the delay (the time a packet is delayed due to queueing). These plots are plotted the same way as was done in [3], with the addition of plotting the delay. In the *Paths used* graph, the total amount of paths used throughout the run can be seen. Eight is the maximum number of paths. In models with both a predictive and a reactive control, the reactive controls are marked as a red dot, so it is possible to see how often the reactive control was triggered. The *Capacity* graph shows the available capacity by adding up the capacity for all the used paths at a certain time, taking into account if the path is in LoS or not. The



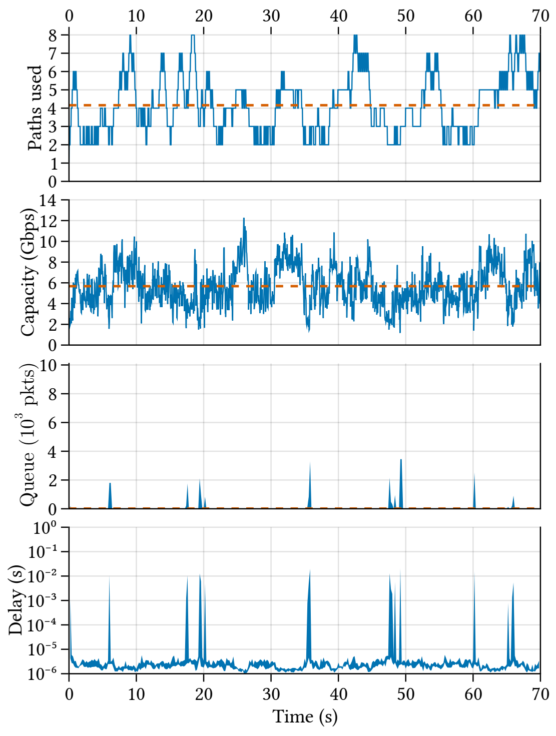
(a) Reactive control



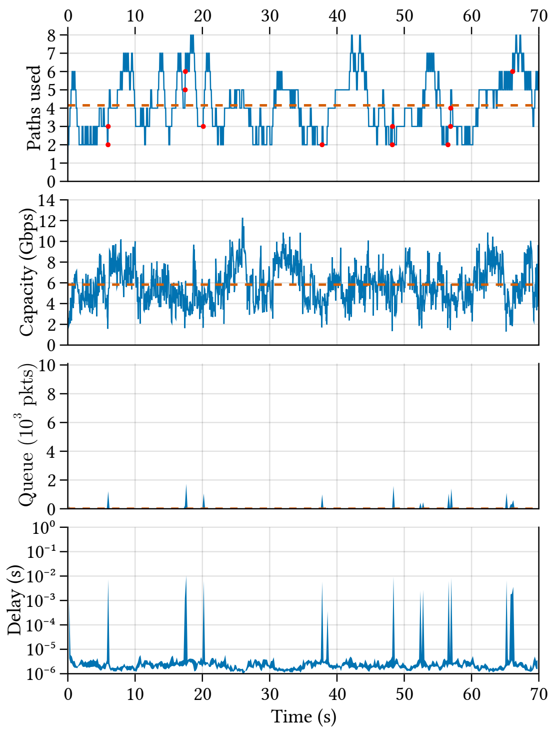
(b) CDF-based prediction



(c) CDF-based prediction with reactive control



(d) FPT-based prediction



(e) FPT-based prediction with reactive control

Figure 4.1: Running the simulation with different path management schemes.

Queue and the *Delay* graphs are plotted as a band for the range (min and max) of the queue size and delay experienced by the system over 200 ms intervals. Packet loss is marked as black dots in the *Queue* and the *Delay* plots. The average is plotted as a dashed red horizontal line.

As expected, the purely reactive control (see Figure 4.1a) has the highest queue levels and the most delay out of the five algorithms. This is because the reactive control cannot react fast enough, thus the queue is allowed to increase, and the delay goes up. Next, the improvements when adding a reactive control to a predictive-based algorithm are clearly seen. Comparing the CDF-based algorithms, the queue spikes that come at the end of the simulation run in the purely predictive model (see Figure 4.1b) are almost completely gone when adding a reactive control (see Figure 4.1c). This performance boost is also visible in the new FPT-based algorithm, where the queue is overall smaller, and the delay spikes are not as high.

Figure 4.1c and Figure 4.1e shows how often a path change was triggered due to a reactive control. This is marked as a red dot in the *Paths used* plot. A path change was not often triggered by a reactive control. Twelve times for both the CDF- and the FPT-based model. Compared to how often a predictive control is triggered in the purely predictive ones – which is around 470 times – this is good. It reveals that the purely predictive-based models can adequately capture the queue so that the reactive control does not need to correct things that often.

4.3.2 The more steady performance

After the first impression of the performance of the two new models, a new test is conducted to find out how the models perform after running them 100 times in the simulator. For each simulation run, the packet delay for each packet is collected to increment the corresponding bin in an array to ultimately create a histogram, which is used to create a CDF of the delay distribution. The packet delay is the time a packet spends in the proxy, from arriving in the proxy to departing. The most prominent task of the proxy is to ensure that the high data rates passing through do not get delayed while being in the proxy; thus the time each packet spends in the proxy should be as little as possible. From Figure 4.1d to Figure 4.1e the queue is decreased, but the difference in delay is not that prominent. Running the simulation 100 times for each model will better show the difference between the different models.

Figure 4.2 show the results of this experiment. The delay has significantly decreased when going from a purely reactive approach to a proactive one. As for the four predictive models, the difference between them is fairly small. This is because the queue is mostly empty, as seen previously, thereby preventing the delay from becoming very high. When using the reactive control, the delay is much more notable, as the draining of the queue goes slower than with the other models. This results in a much higher delay than the predictive control models have.

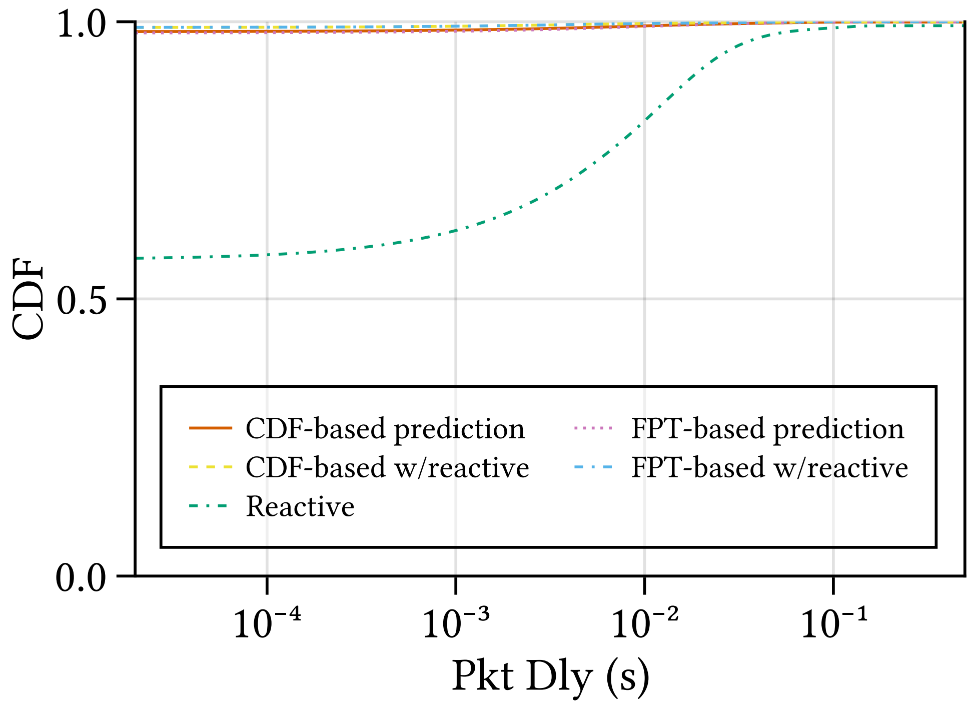


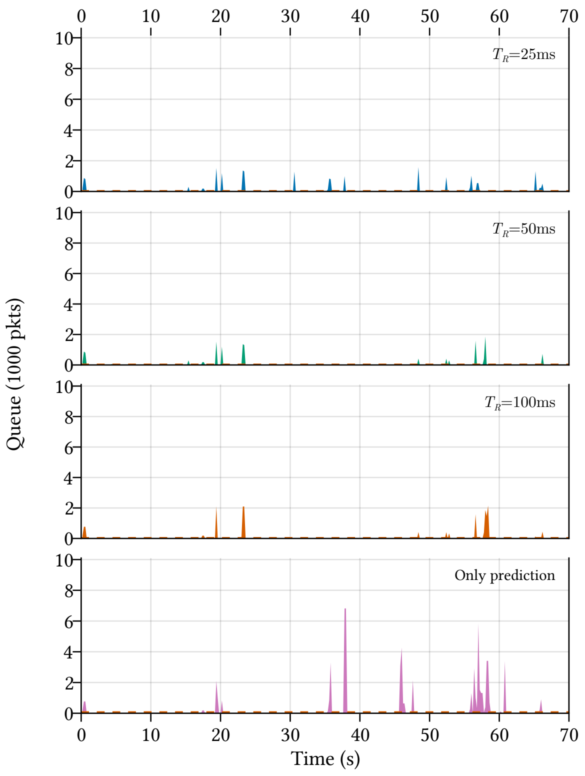
Figure 4.2: Delay CDF from 100 simulations

One important aspect when implementing a reactive control mechanism into the predictive control is limiting when a reactive control is allowed to be triggered. This is limited by the reactive control interval, T_R (see section 3.3). Next, this value is explored to see how it affects the results.

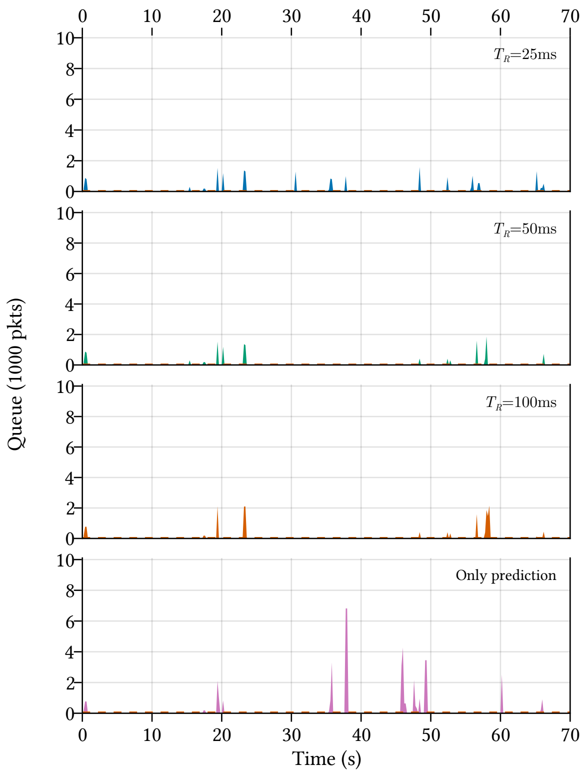
4.3.3 What should the reactive control interval be?

The reactive control interval, T_R, is a value that is important to set correctly, as it is used to tell how soon after a path change, another path change can be initiated because of reactive control. If T_R is too big, the benefit of adding the reactive control to a predictive-based algorithm will go away. If the predictive control is triggered every 150 ms, the reactive control will never be triggered if the reactive control interval is higher than this. Too small T_R is not good either. As simulated in the simulator, a path change does take some time. If T_R is smaller than the times it takes to change the number of paths, a new reactive control will have time to be triggered before a previous path change is set into effect. This is not something an operator would want and maybe do not accept altogether.

Three different values of T_R are chosen: 25 ms, 50 ms, and 100 ms. They are all higher than the time it takes to change paths (20 ms in the simulator) and lower than how often the predictive control is triggered (150 ms). First, each algorithm is run once, and the queue size and the delay are collected. Figure 4.3 shows the queue, and Figure 4.4



(a) CDF-based prediction



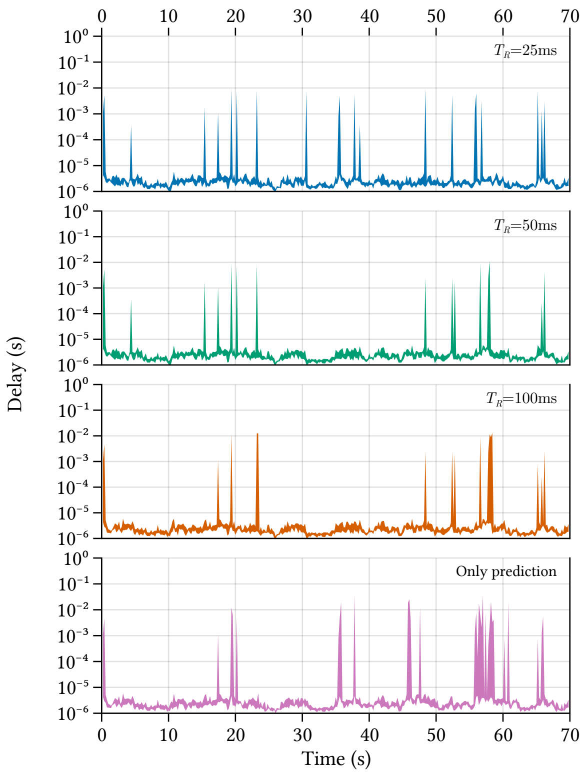
(b) FPT-based prediction

Figure 4.3: The queue occupancy during one run of the simulation. In the three upper plots a predictive control with reactive control is used with different values of T_R. In the bottom plot, purely predictive-based control is used.

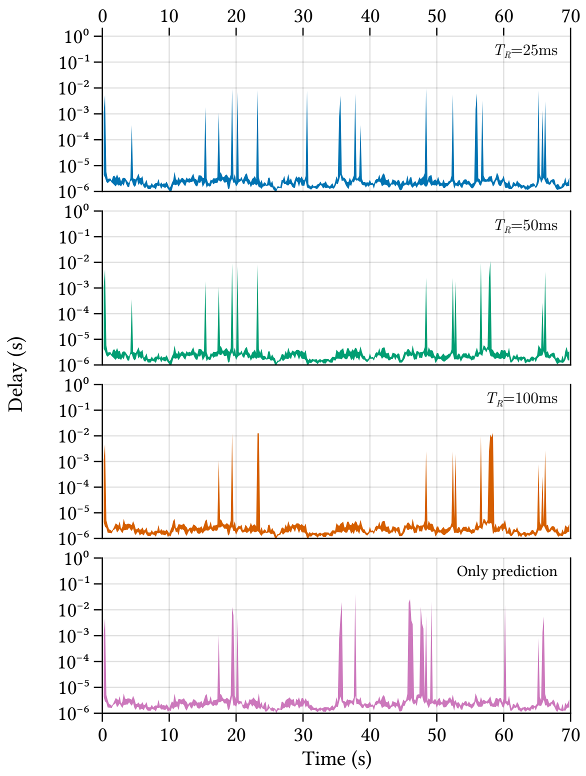
shows the delay after one run of the simulation with different values of T_R. In Figure 4.3a and Figure 4.3b the queue clearly increases as T_R increases. This is expected, as it takes a longer time before a path gets added as T_R gets bigger, thus more time for the queue to fill up. These figures also portray that the queue has spikes in different places. This is because the following predictive controls get shifted some milliseconds forward whenever a reactive control is triggered. This means that the predictions will be a little different than the original. Thus the queue will spike at different times.

In Figure 4.4 shows the delay throughout one simulation run. In Figure 4.4a, the delay seen between 50 and 60 seconds after the start of the simulation in the purely predictive based control in the bottom plot is strongly reduced when adding a reactive control interval. What is not very clear from these plots is how the performance improves, in terms of smaller delay, as T_R decreases. There is a minor increase in the delay as T_R increases, but to highlight this, the simulations are rerun 100 times to make the difference more noticeable.

The same simulation is run 100 times for each of the values of T_R = {25, 50, 100} ms. Figure 4.5 shows the result of this. Note that the figure



(a) CDF-based prediction



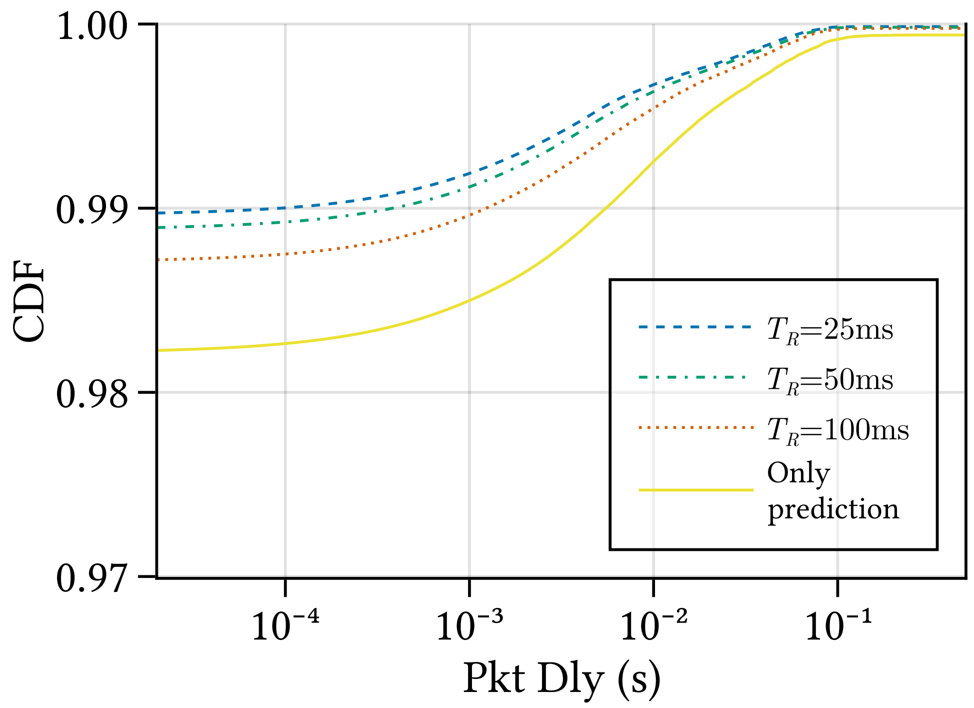
(b) FPT-based prediction

Figure 4.4: The delay during one run of the simulation. In the three upper plots a predictive control with reactive control is used with different values of T_R. In the bottom plot, purely predictive-based control is used.

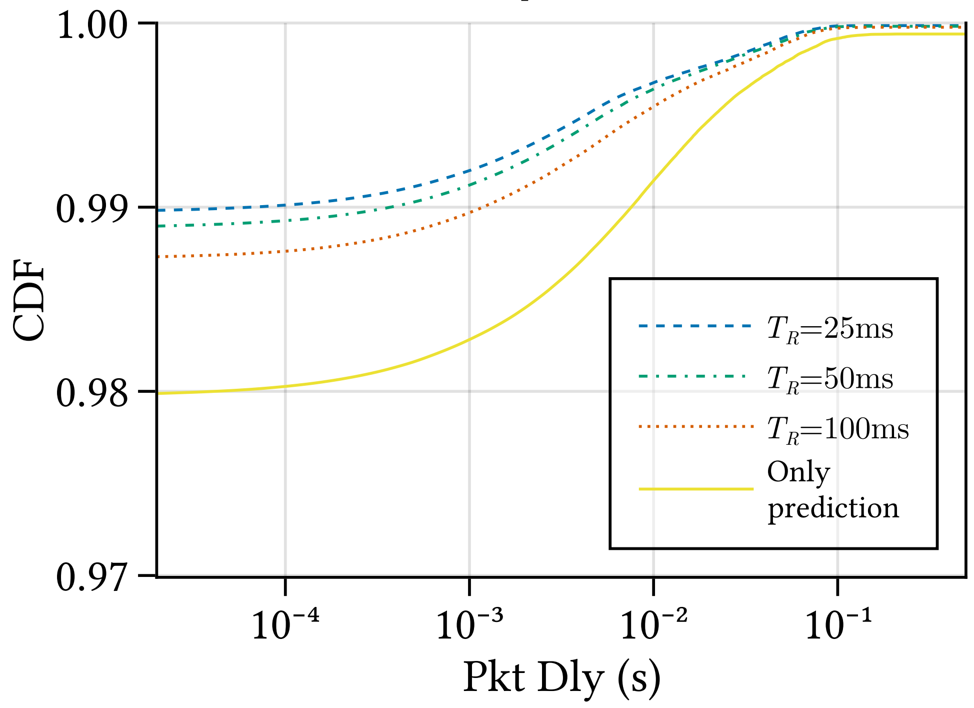
is zoomed in on the very top of the CDF because the delay, which is being minimized, is very low (see Figure 4.2), so to see the difference among them, the scale has to be changed. The effect that T_R have on the delay is clearly seen when using both the CDF-based control in Figure 4.5a and when using the FPT-based control in Figure 4.5b.

If there is not any packet loss, then the CDF of the delay will reach one. However a small amount of packet loss is detected in Figure 4.5a and Figure 4.5b for the purely predictive based models when zoomed in on the top 3% of the CDF. This packet loss is reduced when adding a reactive control to the purely predictive-based models.

Table 4.1 shows how many times the reactive control, on average, was triggered for the different values of T_R. When using a purely predictive-based control method (using either CDF- or FPT-based prediction), the number of times the predictive control is triggered is around 470 times. This means that during the simulation time of 70 seconds, a path change can potentially happen 470 times. How often a path change does happen depends on how the queue looks. If the queue is steady and the capacity of the paths in use does not change, a path will neither be added nor removed. That a reactive control does not



(a) CDF-based prediction



(b) FPT-based prediction

Figure 4.5: Delay CDF from 100 simulations, testing different values of T_R and comparing them to a purely predictive one.

Table 4.1: Average number of path changes caused by a reactive control after 100 runs of the simulation. In the purely predictive control models, around 470 predictive control actions are performed in each run.

Prediction	Reactive control interval T_R		
	25 ms	50 ms	100 ms
CDF-based	9.14	6.72	5.43
FPT-based	8.86	6.59	5.26

trigger a path change frequently is a good thing. This tells us that the predictions are mostly correct, and the reactive control does not have to react to abrupt changes in the queue. What is also possible to see from Table 4.1 is that as T_R increases, the number of path changes triggered by a reactive control decreases.

4.4 Beyond simulations

Testing the models proposed in this thesis in the simulator shows that incorporating a reactive control backup into a predictive-based model yields better performance. Parts of the results from this simulation study are reported in section 5.4 of [4].

The next step is to test the new models in a more realistic scenario. Since mmWave is not widely deployed yet, testing it in a real-world setup is impossible. The next best thing is to use an emulator. An emulator is like a simulator, except where you can control everything in a simulator, an emulator has some elements that you cannot control. This makes an emulator more realistic; thus the results from comparing them will be of higher significance.

In the simulations, only path management is considered. A more realistic scenario would be emulated when going over to an emulator. This means that more than just path management must be included. A full working proxy must be implemented, including path management and packet scheduling. In addition, costs that need to be considered in the simulation (e.g. calculation time for the predictive control) are automatically included in an emulation.

Chapter 5

Design and Implementation of the proxy

After implementing the two new models in a simulated environment, where they could be tested and compared to other path management models, testing them in a more realistic scenario is necessary. This is because a simulation cannot validate the models, only instantiate them. Before running any tests, the models must be converted from the simulated environment to an emulated one. A proxy needs to be developed to test the path manager models in an emulated environment. In this section, the design of the proxy and how it is implemented are explained.

This chapter will answer two of the research questions defined in this thesis:

- *How can a proxy be implemented to work in a mmWave network?*
- *Is the Julia programming language an adequate choice for implementing a real-world proxy?*

5.1 The Design of the Proxy

Parallelization is an essential step in developing a high-performance proxy that is to be used in a demanding network with very high data rates. Doing the different tasks sequentially will take too long. The proxy is divided into three main modules, which is illustrated in Figure 5.1 with the main task that each module has. They run concurrently and are loosely connected.

The first main module is packet handling, which deals with incoming packets. The second is path management, which deals with the management of the paths, i.e. deciding which path should be operative. Lastly, the module responsible for handling the outgoing packets is called the packet scheduling module.

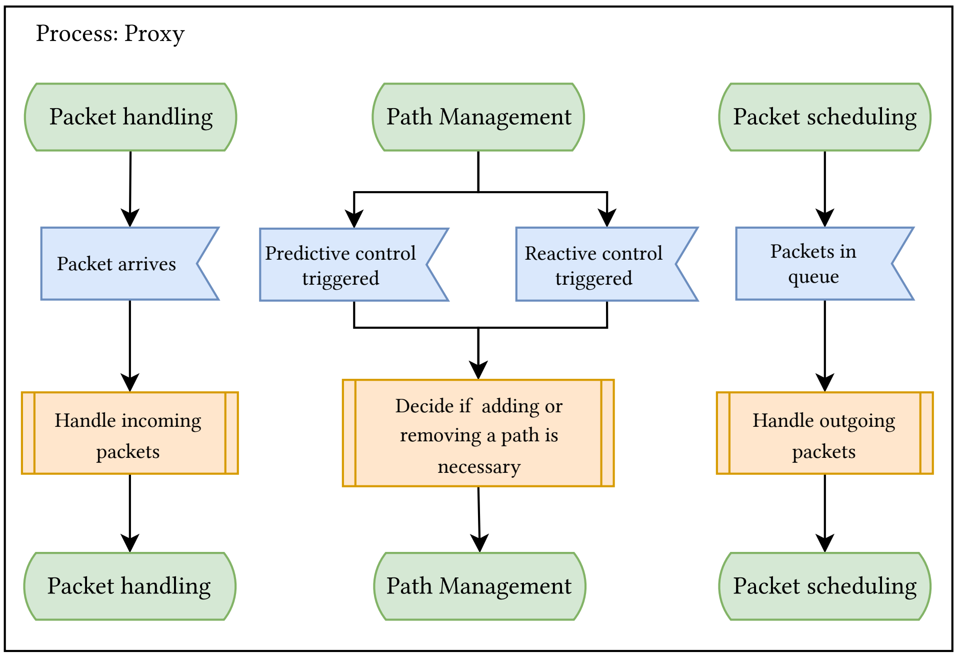


Figure 5.1: An SDL [29] of the proxy showing an overview of the three modules it consists of, with their main task.

5.1.1 Packet handling

The first task that the proxy consists of is packet handling. Figure 5.2 is an SDL of the packet handling module. The main job of this module is to receive incoming packets from the internet and place them in a First-In First-Out (FIFO) queue. This is a type of queue where the first element that arrives is the first element that leaves the queue.

When an incoming packet arrives, the queue is inspected to ensure that it is not full. If the queue is full, the packet gets dropped, and the task goes back to being idle while waiting for a new packet. If there is room for a new packet in the queue, the packet gets added, and the packet scheduling module is notified that a packet has arrived that needs to be sent to the receiver. Next, if adding a packet into the queue made the queue level reach a high threshold, *Q_HT*, then a reactive control will be triggered in the path management module. Lastly, the module will go back to being idle while waiting for a new packet to arrive.

5.1.2 Packet scheduling

Packet scheduling is the module responsible for sending the packets in the queue to the receiver. As packet scheduling is not the scope of this thesis, no scheduling algorithm is used for selecting which path a packet should be sent over. Whenever a packet is in the queue, one of the available paths will take it out and transmit it to the receiver. An available path is a path that is operational and is not busy sending

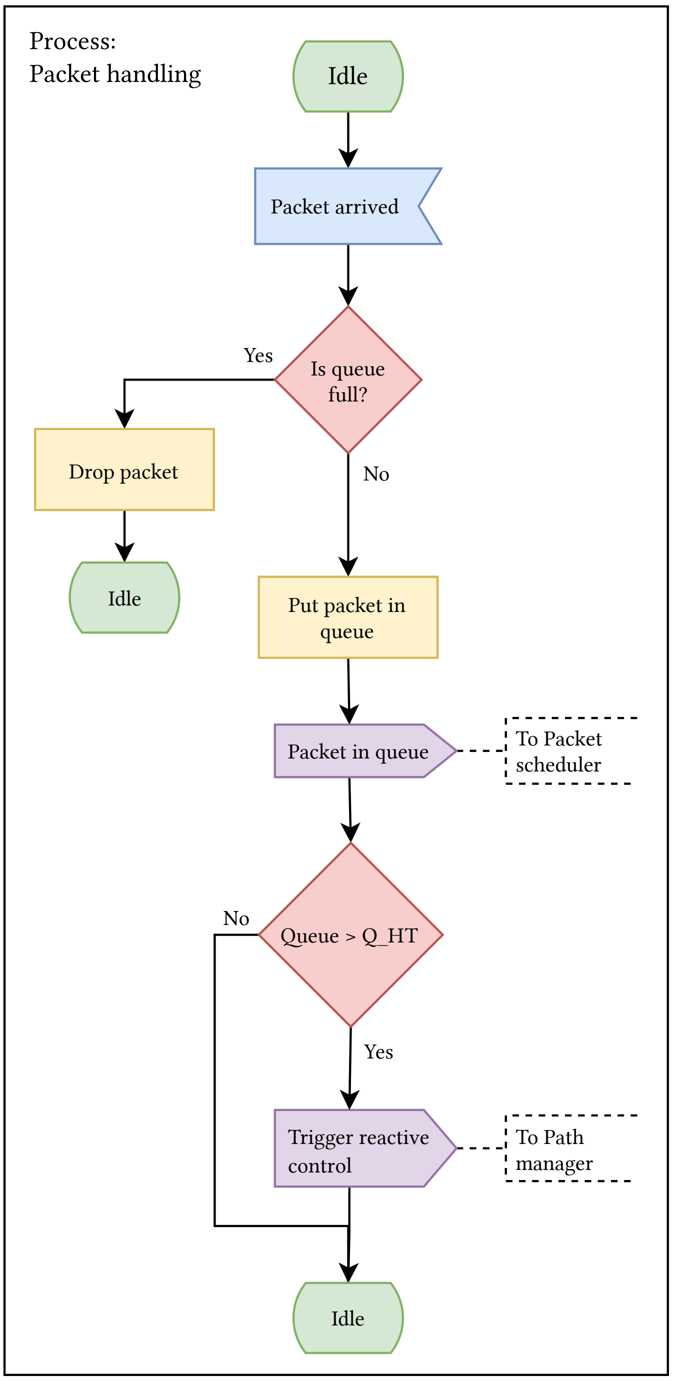


Figure 5.2: SDL [29] of packet handling

packets.

That no specific packet scheduler is used in the proxy, but rather each available path being greedy by taking out the first packet they can, can give a bad performance. For example, packets can be delivered out of order at the receiver. Furthermore, packets can be transmitted over a path that is in NLoS, which will increase the possibility of packet loss.

5.1.3 Path management

The path management module is responsible for maintaining the paths. This means that it should decide how many paths should be operative to keep the desired QoS. This is done in two ways, illustrated in Figure 5.3.

The first approach to maintaining the paths to achieve a desired QoS is the predictive approach. The predictive control is triggered every 150 ms, as in the simulator. Once triggered, a new predictive control is scheduled. This ensures that the next predictive control happens 150 ms after the previous predictive control was triggered. After that, the calculations start, where it predicts how the queue will look in the next 200 ms, called the time horizon. What type of calculation is performed is based on the control method used. It can either be CDF-based or FPT-based, as described in chapter 3. When the calculations are complete, a check is performed to ensure that a reactive control did not happen while the calculations ran. This will ensure that a new path change does not occur too soon after a previous path change.

The results of the calculations can be to add a path, remove a path, or do nothing. If it is to remove a path, a check is performed to ensure that removing a path is OK. This check includes running the prediction again, but with a path less. This check is only performed if less than five paths are in use. The fewer paths there are, the more critical it is to remove one. A signal is sent to the operator to tell it to remove a path if it is OK to remove a path. Suppose the decision was to add a path, a signal is sent to the operator right away, telling it to add a new path.

The second way of changing the number of paths is if a reactive control is triggered. Reactive control is triggered when the packet handling module signals to the path management module that a reactive control is to be triggered. This will happen if the queue reaches a predefined threshold. When triggered, the first thing that happens is that a new predictive control is scheduled in 150 ms. Then a signal is sent to the operator to request the addition of a path. If there are more paths to add, the operator will add a path.

5.1.4 Additional components: Operator and LoS/NLoS switcher

The three main modules described so far are what the proxy will consist of in a real system. However, this proxy will not be tested in a real system but in an emulated one. Two additional components are

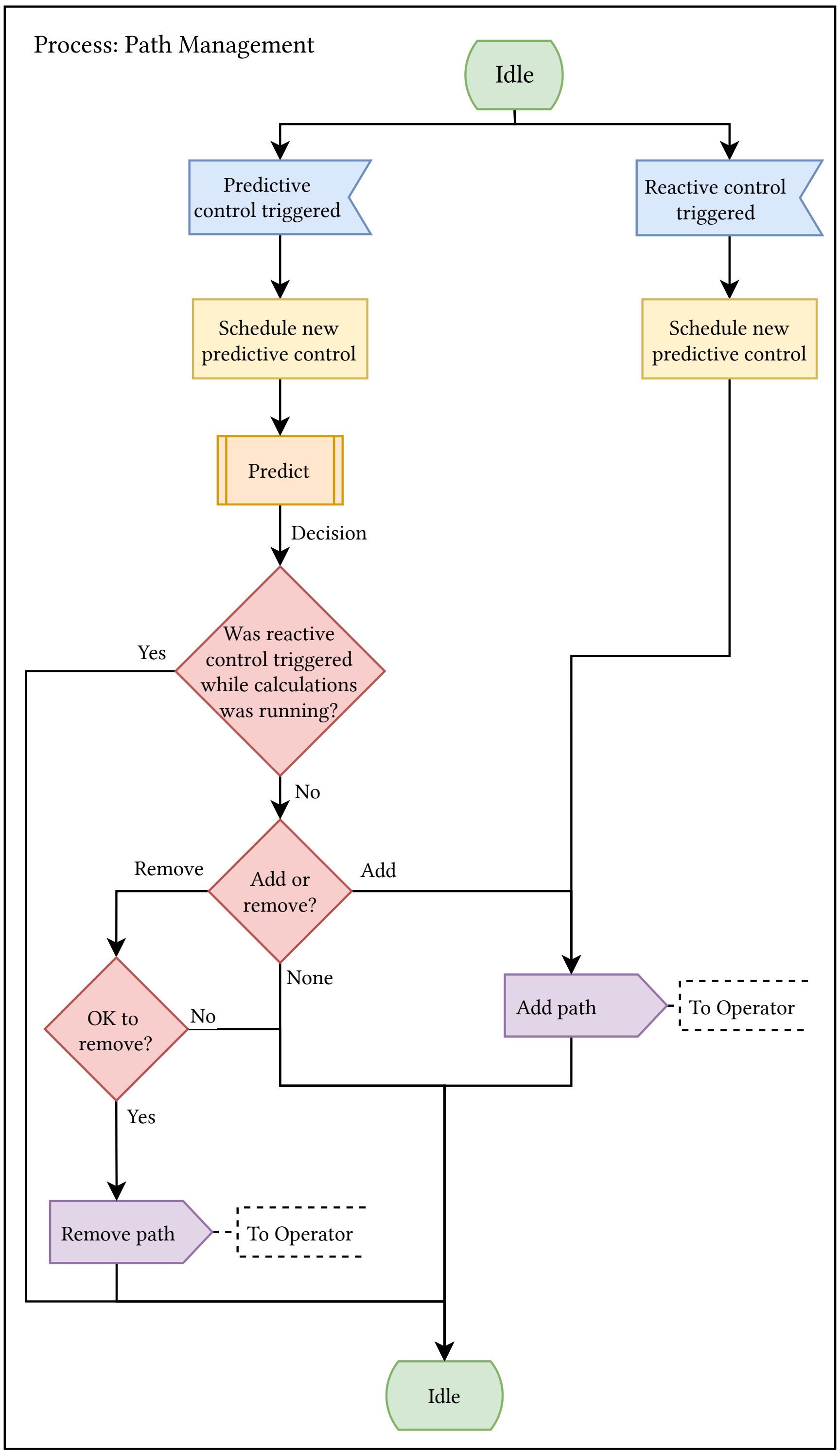

Process: Path Management
Idle
Predictive control triggered
Reactive control triggered
Schedule new predictive control
Schedule new predictive control
Predict
Decision
Was reactive control triggered while calculations was running?
Yes
No
Add or remove?
Remove
Add
None
OK to remove?
No
Yes
Add path
To Operator
Remove path
To Operator
Idle

Figure 5.3: SDL [29] of path management

introduced to deal with things that are not the proxy's responsibility: an operator and LoS/NLoS switcher of the paths.

The operator and the LoS/NLoS component are not part of the proxy in a real system, but these modules are necessary to keep the system as realistic as possible in an emulated one. The operator has the function of adding/removing a path according to what the path manager decides, and the LoS/NLoS component changes the capacities for the different paths to emulate a mmWave network.

Operator

The operator is responsible for adding and removing paths. It receives a signal from the path management module that a path is either to be added or removed. When a signal arrives, the operator needs to first wait *PathChangeDelay* time before doing anything. This is included to emulate a real system where adding/removing a path takes some time. After waiting, the operator finds out if the signal it received was to add or remove a path and acts accordingly.

LoS/NLoS switcher

The LoS/NLoS switcher changes the capacity for the different paths to simulate a moving person in an urban environment where the LoS to the different base stations change as the person moves around. Both the shadowing fading effect and the LoS/NLoS dynamics of the paths are simulated in the simulator in chapter 4. This is not done in the emulator. Only the LoS/NLoS dynamics of the paths are emulated since emulating the shadowing fading effect will imply changing the capacity of the paths too often. On average, emulating the LoS/NLoS dynamics of the paths will mean changing the capacity every third second.

5.2 Implementation

The proxy is, as the simulator, written in Julia 1.7 [31]. This was chosen for two distinct reasons. The first is that since the simulator is already written in Julia, transferring the code used in the simulator to the proxy is easier. Second, Julia is a programming language that is supposed to be fast and efficient, which is suitable for the proxy since it should run fast. This section explains how the proxy is implemented with regard to the design.

5.2.1 Parallelization

The proxy consists of, in total, five modules that must run concurrently: *packet handling*, *path management*, *packet scheduling*, *operator* and *LoS/NLoS switcher*. This makes parallelization an essential step in implementing the proxy. In Julia, four different parallelization mechanisms exist: coroutines, multi-threading, distributed computing, and

GPU computing [32]. Coroutines are not strictly parallel computing but allow suspending and resuming computation for I/O, event handling, etc. Julia's multi-threading allows to schedule tasks on multiple threads or CPU cores concurrently, where the tasks share memory. Distributed computing can start numerous processes, where each process has its own memory space, and finally, GPU computing allows the programmer to run Julia code natively on GPUs.

Initially, distributed computing was used in the proxy, where each module would run on different processes with different memory spaces. Having separate memory space is easier to work with, and initially, there was not much that needed to be shared between the modules. But, as the system's complexity evolved, what needed to be shared between the different modules also changed. There are some data structures in Julia that support multi-processing, like SharedArray and RemoteChannel [33]. However, they do come with their limitations. For example, SharedArray can only store elements of bit type. This limits to mainly storing numbers, which is too limiting to be used in the proxy.

After initially trying to implement the proxy using distributed computing, using multiple threads was chosen instead, as I found it easier for this proof-of-concept prototype. As mentioned, threads share memory space; thus, there are no limitations to what type of data structure that can be used. However, another problem arises when the memory space is shared: thread safety. Keeping code thread-safe means ensuring that only one thread can modify or read the shared data at a time. Suppose multiple threads access the same data element, where one writes, and another reads; unexpected things can happen. Locks can be used to ensure that only one thread can access an element at a time. Locking and unlocking is a way to ensure that reading and writing to an element are done securely.

There are different ways of solving the locking of elements. One solution is to have one global lock that each thread must try to acquire when it wants to read/write an element. This, however, has the effect of limiting parallel execution and can slow down the system. If one thread wants to use the lock to retrieve the information about a path, it will block another thread that may wish to access a different element. This is unnecessary blocking and can give a lower performance. Another solution, which is the one that is in use in the proxy, is to have a different lock for each element that different threads may modify. This will reduce the blocking that could happen when using one global lock. However, it is important to remember that locking and unlocking are not trivial, with regard to the time it takes to acquire a lock, and should be kept at a minimum.

The different threads in the proxy

The proxy consists of nine main threads that run concurrently: one thread to handle incoming packets, four threads for handling outgoing packets (one thread per path), one thread for the path management, one thread for the operator, and one thread for the LoS/NLoS switcher. Some additional threads are also used, which will be explained later.

To start one of the modules in a new thread, the build-in macro `@spawn` [34], which comes from the Threads module, can be used. For example: `@Threads.spawn packet_handling()`. Here the packet handling module will start in one of the available idle threads. The disadvantage of using `@spawn` is that there is no guarantee that a new thread will be used. An already working thread can be picked if it is incidentally idle. To avoid two modules (e.g. packet handling and path management) running on the same thread, ThreadPools is used [35]. The macro `@tspawnat` in ThreadPools can be used in the same way as `@spawn` but have one more argument: the thread id, which specifies which thread the task should start on [36]. An example of how it is used: `@tspawnat 2 packet_handling()`. When this is called, the packet handling module will start on the thread with id 2. This can then be used to start the different modules on different threads and ensure that they run concurrently.

When using ThreadPools, the scheduling is manually done since the different modules are forced to run on the assigned thread. However, this is not done for all of the threads in use in the proxy. Every time the calculations for the predictions are made, the work will be distributed on four threads by the math kernel library (MKL) [37]. Which threads that are selected are not something that is chosen beforehand, thus a thread that is assigned to one of the modules can be selected so that that module can not do its work while the calculations are running. How significant a disadvantage this is depends on how long time it takes to finish the calculations.

5.2.2 A shared data structure

One shared struct is used to store information about the system, called PathSystem. This is shared between all the threads, i.e. the modules. In PathSystem, the queue, the list of paths, and several variables are stored.

The list of paths is a Vector that stores elements of the type Path. A Path is a structure where all information about a path is stored. This includes to-address, from-address, the sockets used to send packets over, if the path is active or not, if it is in LoS or not, the available capacity, and the average time in LoS and NLoS. The addresses are stored as a `Sockets.InetAddr`, which is a type that can hold both the IP address and the port number. The socket used to send packets over is of type UDP.

PathSystem also stores values like which control method is in use (eg. reactive control, CDF-based prediction, CDF-based prediction with reactive control, etc.). The thresholds used for the queue are also stored: *Q_HT*, *Q_HT_prob*, *Q_LT*, *Q_LT_prob*. They are used to decide if it is necessary to add a path, remove a path, or do nothing. *t_IC* is the last time a change was initiated, and *PC_token* is the next predictive control token.

5.2.3 The queue

The data structure used for the queue has to support multiple threads adding and removing elements from the queue at a fast rate. Two different data structures were tested: Julia's `Channel` [38] and DualLinkedConcurrentRingQueue [39] from the ConcurrentCollections package.

A `Channel` is a FIFO queue, which is optimized for multi-threading since multiple threads can read and write to it. It is fast and efficient, making it a good choice for the queue where it is expected that the two threads will put and take elements in and out of the queue at a high rate. However, when using the `Channel` data structure, it was found that it did not perform as required. Every time something is put in the queue or something is taken out, a lock is needed. When looking at the source code for Julia's `Channel`, the lock is held unnecessarily long, making it slow down the process of adding/removing packets from the queue. It is stated in the documentation that a `Channel` is efficient at adding and removing elements to and from the queue, but not as efficient as needed for the proxy.

DualLinkedConcurrentRingQueue (DLCR queue) is a new data structure that implements an almost non-blocking queue [40]. By reducing the number of locks needed and reducing the time a lock has to be held for, adding and removing elements from the queue can be done quicker. A difference between a DLCR queue and Julia's `Channel` is that the DLCR queue is boundless. When creating a `Channel`, how many elements it maximum can have can be specified. This is not something that can be specified when creating a DualLinkedConcurrentRingQueue.

Another difference is that in Julia's `Channel` how many elements the `Channel` currently has can be accessed by a function call. This is not supported in a DLCR queue. An atomic counter [41] is used to always know how many packets there are in the DLCR queue. This means that every time a packet gets put in the queue or removed from the queue, a lock has to be used to increment/decrement the atomic queue length counter. However, holding a lock to only add or subtract a number by one does not take a long time.

5.2.4 Message passing between the modules

Julia's `Channel` [38] is used to pass signals between the different threads in the proxy. Using a `Channel` as the proxy's queue was not fast enough, since it is expected that taking elements in and out of the queue is done at a very fast rate. However, a `Channel` is very suitable as a way to send signals between different threads, where elements will not be added and removed at a high rate. Furthermore, a `Channel` is thread-safe and optimized to work with threads.

One example is the signal to the operator. Whenever a path change is initiated, the operator has to be notified that it should add or remove a path. This is done by placing an action signal (add or remove) in a `Channel` by the path manager thread, which is then read by the operator thread.

5.2.5 Accurate timing in the proxy

For all of the different modules that the proxy consists of, all of them have to sleep either periodically or sporadically. One way of sleeping in Julia is to use `sleep()` [42]. However, the minimum sleep time this function accepts is one millisecond. Furthermore, when testing this sleeper, the accuracy was not good, as it often slept for a longer time than it should. Another function that can be used is `systemsleep()` [43]. This function blocks the Julia thread that it is running on, thereby preventing other tasks from using this thread while it is idle. This may give a lower performance since one thread can only be used for one task, consequently having one available thread less to use for the proxy. Moreover, the accuracy of this sleeper did not suffice to be used in the proxy.

Both `sleep()` and `systemsleep()` has too high granularity (1 ms), which prevents sleeping for microseconds at a time with high accuracy. Linux has a function called `clock_nanosleep` [44], which allows the calling thread to sleep with nanoseconds precision. It can be accessed using a system call to libc [45]. As the proxy aims for microsecond precision, this is more than adequate to be used in the proxy. To call a C function in Julia, a **`ccall`** to the appropriate C function, in this case `clock_nanosleep`, has to be made. This C call is implemented in a function called `nanosleep!()`, which takes two arguments: a *timespec* structure used as starting time and how many nanoseconds from this starting time the sleep should last. For measuring the time a monotonic clock is used, which is a clock that measures the time since some unspecified point in the past and will not change after system startup.

`nanosleep!()` uses absolute time when measuring how long the sleep should last. `nanosleep!()` works as follows. First, a timespec structure, `ts`, is made, and what the time is now, found by looking up the monotonic clock of the system, is filled into the timespec structure. Then, a call to `nanosleep!(ts, sleep_time)` is made, where `ts` is the time structure created and `sleep_time` is how long the sleep should last. `ts` will then be added with `sleep_time`, and when the time is more than `ts+sleep_time` the sleep should quit.

Repeated calls to `nanosleep!(ts, sleep_time)` will continue to add `sleep_time` to `ts`. This means that if a call to `nanosleep!()` returns late one time, the next time `nanosleep!(ts, sleep_time)` is called, the sleep will not last `sleep_time` amount of time, but less. This means that if `nanosleep!(ts, sleep_time)` is called repeatedly, the sleep time will on average be `sleep_time`, but sometime it will be more and sometime less.

This behavior can be useful in some cases and useless at other times. An example of where this behavior is useful is when sending packets to the proxy, where the goal is to send at a certain rate. If the goal is to send a packet every second, sending two packets with half a second interval and two other packets with two seconds intervals do not make a huge difference. As long as the average interval between the packets is one second, it is fine. However, there is a use case in the proxy where the sleep function should not expire before `sleep_time` amount of time has past, and that is in the path manager. The path manager should trigger the predictive control every 150 ms, but not less than this. To ensure that a call to `nanosleep!(ts, sleep_time)` does not sleep less than `sleep_time` amount of time, finding the time by checking the monotonic clock and setting `ts` accordingly before the call to `nanosleep!()` needs to be made. This will prevent that a late sleep previously does not affect the following sleep by sleeping less.

Using `nanosleep!()` in the proxy every time microseconds accuracy is needed gives good results, in terms of the accuracy of the sleep. However, calling a blocking C function is problematic because of how multithreading is done in Julia 1.7. Calling Julia's `sleep()` will handle multithreading nicely by allowing the thread where the `sleep()` call was made on to be used by other tasks in the running program. Calling `nanosleep!()` will not be handled the same way, as it is a blocking C function that is being called. Furthermore, Julia uses a scheduling algorithm that is called static [46], which means that a new task is iteratively assigned to one of the threads, regardless of the workload. This means that if many threads are blocked because of a call to `nanosleep!()`, the possibility of a task being scheduled on one of these threads increases. This can block the whole system so that nothing is able to run.

A way to come around this is to only call `nanosleep!()` from one thread. David Hayes, one of the supervisors for this thesis, implemented `TimeQueue`, which is a priority queue, that can be used by tasks to sleep. A new thread is introduced, that will run `TimeQueue`, so that only this thread will be calling `nanosleep!()`. `TimeQueue` has an ingoing queue where it accepts tasks that want to sleep and multiple outgoing channels to notify a task that a sleep is done. The idea is that every 50 microseconds (will use a while-loop with `nanosleep!()` to sleep for 50 microseconds at a time), it will check the ingoing queue (uses a `ConcurrentDict` [47] from ConcurrentCollections for this, where the key is the time the sleep should finish and the value is the channel that is used to notify the task that the sleep is finished) to see if any tasks are finished with their sleep. If any are, they will be notified by a signal sent via a `Channel`, so that they can wake up and continue their execution.

Whenever a module needs to sleep it will call the blocking function `ClockedNanoSleep()`, which takes four arguments: the priority queue for `TimeQueue` that is shared between all tasks, a `Channel` that is specific for each task which is used to signal that a sleep is done, and the same two arguments passed to `nanosleep!()` (a *timespec* structure used as starting time and how many nanoseconds from this starting time the sleep should last). By using a `Channel` in the sleeping task to wait to wake up, a nice blocking will be done, so that other tasks are able to run on that thread. However, the accuracy of this sleeper is based on how quick the task that is waiting for a sleep to finish will be scheduled on one of the threads when a signal is sent to its `Channel` that a sleep is done.

Furthermore, since the `TimeQueue` has to traverse all elements put in its `ConcurrentDict` every 50 microseconds, the timing might be off if it has to loop through many elements, i.e. many tasks want to sleep at the same time. This will however sort itself out as explained earlier because an absolute time is used so if the time `ts+sleep_time`, which are arguments sent to `ClockedNanoSleep()`, is more than the actual time, then it will return immediately without sleeping `sleep_time` first. A more sophisticated priority queue would make this more efficient but was not currently available.

5.2.6 Path management

Every time a predictive control is triggered, the Markov model will be solved by running various calculations. What type of calculations that is done depends on the chosen model, i.e. if it is CDF-based or FPT-based prediction. After the model has been solved, P_H and P_L are returned. These numbers reflect whether the prediction resulted in the desire to add or remove a path. If P_H is less than $P\_InSufficient$, adding a path is necessary. If P_L is more than $P\_MorethanSufficient$, then removing a path is necessary. When adding a path, a signal is sent immediately to the operator to tell it to add a path.

When removing a path, and the number of paths is higher than five, a signal is sent to the operator so that a path can be removed. If the number of paths is five or less a check needs to be performed before signaling the operator, to ensure that removing a path is fine. This is because the impact of removing a path will be higher the fewer paths there are. The check includes of rerunning the calculations with a path less (without actually removing it). If this results in the desire to add a path, then removing a path was not a good idea and nothing is done.

To track when the last control was done, t_IC is used so that t_IC is set to *now* each time a control (predictive or reactive) is performed. This will prevent two controls from being triggered too soon after each other. Predictive control is triggered every 150 ms, given that no reactive control is triggered. A reactive control can be triggered no sooner than the reactive control interval, T_R, allows. This is set to be 50 ms in the proxy. Whenever a reactive control is triggered, t_IC will be updated. This will affect the predictive control in that it has to be shifted some time forward so that there are 150 ms between the reactive control and the next predictive control.

Since the proxy does not know if a reactive control will be triggered or not, a predictive control will be scheduled every 150 ms. However, if a reactive control is triggered, a previously scheduled predictive control will be discarded, and a new predictive control will be scheduled in 150 ms. To know which predictive controls to execute and which one has been discarded, a *PC_token* is used. This is a number that reflects the number given to the latest scheduled predictive control. Whenever a new predictive control is scheduled, *PC_token* will increment by one. When the predictive control is triggered, a check is done to see if that specific predictive control has the same number as *PC_token*. If it is the same, then the predictive control is executed. If it is not the same, a reactive control has happened, which incremented the *PC_token*, and the predictive control comes too soon after, so it gets ignored.

Scheduling a predictive control can be done in various ways. One way is to use a `Timer` [48], which is a built-in function in Julia that will call a callback function when it expires. This can be used to trigger the predictive control in 150 ms. When a predictive control is triggered, a new predictive control is scheduled immediately (see Figure 5.3). This ensures that there are 150 ms between each predictive control. However, while testing the proxy, it was found that the `Timer` in Julia is heavily affected by other things going on in the proxy, consequently delaying the timer, i.e. increasing the time between each predictive control. The predictions try to foresee the queue for the next time horizon, set to 200 ms. This means that the time between each prediction should be no longer than 200 ms, preferably smaller. When using Julia's `Timer`, the time between two predictive controls was sometimes more than 200 ms.

Another way of scheduling the predictive control is to use a while-loop and a sleep function that will sleep for 150 ms before triggering the predictive control. As the accuracy of the sleep function increases, the interval between each predictive control will be closer to 150 ms. As discussed in the previous section, subsection 5.2.5, sleeping in the proxy is done by calling `ClockedNanoSleep()`, which gives accurate sleeping results.

5.2.7 Packet scheduling

Whenever a packet is placed in the queue, it will activate the packet scheduler so that the packet can be transmitted over one of the available paths. Availability in this context is based on two things: 1 – the path has status as active, i.e. it is operational. 2 – the outgoing buffer for the path's socket has less than a couple of packets in it. A while-loop is used so that it first checks if a path is active, then if there is a packet in the queue, and finally if the path is ready to send. If one of the above checks is false, then it will sleep for 100 microseconds by using `ClockedNanoSleep()` before checking again, otherwise, a packet will be taken out of the queue and transmitted to the receiver.

The packet scheduler needs to be optimized to deal with the high number of packets it has to process in a short amount of time. An efficient way of checking the outgoing buffer for the path's socket and sending packets needs to be implemented to decrease the per-packet processing.

Optimizing sending packets

The proxy has to deal with very high data rates, up to several gigabits. To achieve this high rate of incoming packets, the proxy also has to take packets out from the queue and send them over a path quickly. If the queue starts to fill up because the paths are in NLoS, then the proxy has to be able to drain the queue when the paths are in LoS by sending packets faster than it receives them. That means that if the proxy is going to operate at a multigigabit data rate, it should be able to send packets at an even higher rate.

The available capacity will increase as the queue fills up and the proxy adds paths. As the available capacity increase, the possible outgoing speed increase, and the interval in which the packets needs to be sent to achieve that speed will decrease. For example, if the total available capacity is 1 Gbps, then a packet needs to be transmitted every 12 microseconds to achieve that rate. If the total available capacity increases to 3 Gbps, then a packet must be sent every fourth microsecond. To call the send function every fourth microsecond is demanding for the CPUs and will take too long. Instead of sending a packet at a time, one can send a larger package, including multiple packets, and then let the lower layers handle splitting that into the appropriate size that can be sent over the network (in this case 1500 bytes (B), as this is the maximum transmission unit (MTU) size of Ethernet connections).

Generic segmentation offloading (GSO) is a mechanism that enables this by letting the network interface card (NIC) segment the packets into the appropriate size using its own processor, instead of letting the OS use the CPUs to segment the packets. NIC is a hardware component that allows the machine to connect to a network. It has a driver, where the software is installed, and it is this component that can segment packets it receives from the OS. This will allow the proxy to send multiple packets with one call to send, thereby decreasing the CPU's load and increasing the possible outgoing sending rate.

The proxy takes advantage of GSO by taking ten packets out of the queue and storing the content of each packet in a larger packet, which NIC will fragment into appropriate sized packets to be sent to the receiver. However, since the capacity of a path can fluctuate greatly, sending ten packets in one go over a path with low capacity will yield low performance. To mitigate this, ten packets are only taken out of the queue if the path they will be sent over is in LoS. If it is not, only one packet is sent.

While testing the proxy, the thread assigned to sending packets to the receiver ran at around 100% CPU power. This means that it was working as hard as it could. However, the sending rate did not meet the requirements, which means that even though the power of the kernel's CPU was maxed out, the sending rate did not suffice. To mitigate this, a sending thread is used for each of the paths, i.e. four threads are used for the four paths. Each of the four paths has access to the queue, and will greedily take packets out from the queue whenever the paths are available and there are packets in the queue.

When using multiple threads to send packets, using Julia's `send` to send packets is not the best way. Julia's `send` uses Libuv [49] underneath to send packets. However, this does not cope well with multithreading, in that having multiple threads sending things does not speed up the sending. So there is no benefit to adding a thread. To send packets on the different interfaces, a direct call to the C function `sendto` [50] is made. When sending packets this way, the benefit of using multiple threads to send packets is seen, and the proxy is able to send packets faster than it received them, hence it is able to drain the queue.

Checking the buffer

If the buffer for the path's sockets has more than a couple of packets in it, it is probably because the path is in NLoS; thus, sending more packets over this path will likely lead to packet loss (unless the path gets in LoS). To check the buffer for a path's socket, a lookup is done in the /proc/net/udp file, which is a file available on Linux. This file contains information about the UDP sockets that are in use by the system, such as the address and port number of the socket and how many bytes are in the outgoing and incoming queues for the socket. The outgoing queue is the field of interest, and a check is performed to ensure that there are no more bytes than two worth of packets in the queue, i.e. 3000 B, before transmitting any packet over that socket.

However, checking the buffer for every packet is too time-consuming and will prevent the sending rate from reaching gigabit speed. The capacity for a path is known when sending packets over it, and thereby it is also known how quickly the packets will be sent over that path. This means that how long the buffer will be occupied before all the packets transmitted over that path is known and can be used to prevent sending packets over the same path too soon, consequently keeping the buffer of that path's socket low.

Each time a packet, or a collection of packets, is sent over a path, how long the path is estimated to be busy with sending those packets is calculated. Suppose the path has a capacity of 1 Gbps and wants to transmit one packet. In that case, it will take 12 microseconds before the path is ready to send a new packet (the packet will be sent immediately, while the second packet can be sent after 12 microseconds); hence the estimated busy period for that path will be 12 microseconds, plus a few microseconds to compensate for this being a real system, where things do not happen at exact times. The task will be put to sleep, by using `ClockedNanoSleep()`, for the time calculated.

This will prevent other packets to be transmitted over that path.

The busy period only estimates how long the socket's queue will have something in it. To ensure that the estimates are correct, for every 100th packet transmitted over a given path, a lookup in the /proc/net/udp file is done. Since this file reflects how much there is in the queue, a lookup will ensure that the proxy does not keep sending over a path with a full outgoing buffer.

5.2.8 Operator

The operator is responsible for making paths operational and tearing them down. It receives a signal to either add a path or remove a path. After a signal is received, it first needs to wait for *PathChangeDelay* seconds. Changing the number of paths takes some time in a real system, as this tries to emulate. *PathChangeDelay* is set to 20 ms, which is the same as was used in the simulator in chapter 4.

If the operator receives a *add path* signal, then it first needs to check whether there are any paths to add or if all of the available paths are already in use. If there is a path to add, the path with the highest capacity will be activated. If the operator receives the *remove path* signal, it has to check if the proxy uses more than two paths. If the proxy only has one path, removing it will make the proxy have no operational paths. This will give a bad performance and is therefore prevented. If there are at least two operational paths, the path with the lowest capacity will be set to inactive.

5.2.9 LoS/NLoS switcher

As the user moves around, the capacity will fluctuate for each path between the proxy and the receiver. An LoS/NLoS switcher is used to simulate this in the emulator. This module will switch the capacity for each path independently, simulating that a path goes from LoS to NLoS and vice versa. To change the capacity for each of the paths independently, coroutines are used (see subsection 5.2.1). This enables each path to change its capacity autonomously.

Each path in the proxy has two sockets; one is used when the path is in LoS, and the other is used when the path is in NLoS. The reason for using two sockets per path is to be able to emulate a path that has a fluctuating capacity (see subsection 6.2.5 for details on how changing the capacity in the testbed is done). When a path switches from being in LoS to being in NLoS, and vice versa, the socket used to transmit the packets will also change, consequently changing the capacity for that path.

5.2.10 Garbage collection

Garbage collection is the process of freeing unused data objects so that the memory they are using can be reused by the running program

[51]. The garbage collector in Julia uses the *mark and sweep* algorithm, which has two phases: a mark phase and a sweep phase. In the mark phase, every object still in use is marked. In the sweep phase, every object that is not marked will be cleaned, i.e. the memory can be overwritten by someone else.

Every time the garbage collector does a cleanup in Julia, all other threads will be paused while the garbage collector goes through the heap and mark used data objects [52]. If the garbage collector uses a lot of time going through the heap, this will negatively affect the proxy's performance, since all the threads will be blocked while waiting for the garbage collector to finish.

Some garbage collection can be avoided by using pre-allocated data structures rather than dynamically allocated ones. For example, every time a packet is retrieved from the queue, it is temporarily stored in a variable. This variable has been pre-allocated, which will prevent a new memory allocation every time a packet is taken out of the queue.

However, some allocations will be performed during runtime, and the garbage collector will have to clean up every once in a while. As mentioned, every time the garbage collector does a cleanup, all threads are blocked and cannot execute. A smaller cleanup is performed more regularly, every 150 ms, to prevent the garbage collector from doing a big cleanup periodically. This means that the garbage collector will use less time on the cleanup, thus the threads will be blocked for a smaller amount of time, consequently increasing the performance of the proxy.

Chapter 6

Test environments

When developing the proxy, a way to test it is essential. To test the proxy, two different testing environments are used. A virtual testbed, which consists of a virtual machine, is used in the early phases of the proxy development. When the proxy had been tested enough in the virtual testbed, a hardware-based testbed was built so that the proxy could be tested under more realistic conditions. This section describes the virtual testbed and the hardware-based testbed.

6.1 A virtual test bed

While implementing the proxy, a virtual testbed is used for debugging. The testbed consists of a VM, where the proxy run, and a laptop, where the sender and the receiver run. A virtual testbed is used because it is easy to set up, making it easy to start developing the proxy. In addition, the virtual testbed can simulate that the proxy and the sender/receiver are running on different machines, which is helpful while implementing the proxy.

The testbed consists of one laptop and a VirtualBox [53] VM running Manjaro over Arch Linux 64-bit. The proxy runs on the VM, and the sender/receiver runs on the laptop. For one VM in VirtualBox, four different interfaces are available. One interface is used to connect the VM to the internet via NAT [54]. The three other interfaces use host-only mode, which allows the VM to be connected to the host, i.e. the laptop the VM is running on, in an isolated network. Eight interfaces are available between the proxy and the receiver in the simulator and later in the testbed. This is not achievable in the virtual testbed. However, as the VM's goal is to test if the code works, having a smaller number of paths will not affect the testing. In addition to adding network interfaces in the VM, multiple processors are added. This enables the proxy to run on multiple threads, which is an essential part of the design of the proxy as explained in chapter 5.

The laptop that runs the VM runs the sender and the receiver. The sender sends packets to the proxy, while the receiver receives packets from the proxy. Since the sender and the receiver is running on the same machine, the packets will go from the laptop (the sender) to the VM (the proxy) and back again to the laptop (the receiver). Since there are only three interfaces between the laptop and the VM, one of them is used for both sending and receiving data. This means that the proxy has one interface for incoming packets and three interfaces for outgoing packets, where one interface has to both receive and send packets.

Using the virtual testbed is a good approach in the early phases of proxy development. However, as the proxy grew and the number of threads increased, the proxy could no longer run in the virtual testbed. The reason for this is the number of processors available. At the beginning of the proxy development, only three threads were used. Thus four processors were enough. However, when this number increased, the limited number of processors became a problem. At this point, it was also time to set up the hardware-based testbed. The proxy implementation was not completely done when the transition to the hardware-based testbed was made. Nevertheless, the proxy got tested in the virtual testbed, which resolved many bugs under development.

6.2 A hardware-based testbed

After the initial testing using a virtual testbed is done, the proxy is tested in a real testbed, using real hardware. Before any tests could be done on the testbed, it must be built. One of the supervisors for this thesis, David Hayes, helped build the testbed. This section explains all the testbed's components and how they were set up to work for our needs.

6.2.1 The network topology

The main goal of the testbed is to test the path manager used in the proxy. There needs to be a proxy connected to a receiver through multiple connections to test this. Figure 6.1 depicts the testbed, which consists of two machines and a switch between them. The proxy runs in one machine, while the receiver runs in the other machine.

A switch enables multiple ethernet connections between the proxy and the receiver, where each connection will emulate a mmWave link. This is done because the receiver does not have numerous interfaces that the proxy can directly connect to; hence a switch is used where the proxy can have eight connections to the switch, and the switch can set up one high-speed connection to the receiver.

To test the proxy's path manager, it needs to receive packets. This is done by placing the sender in the same machine as the proxy, and then letting the machine generate packets.

6.2.2 Hardware used in the testbed

The testbed will be emulating a mmWave network, which operates with data rates of several gigabits. This makes the demand for the hardware

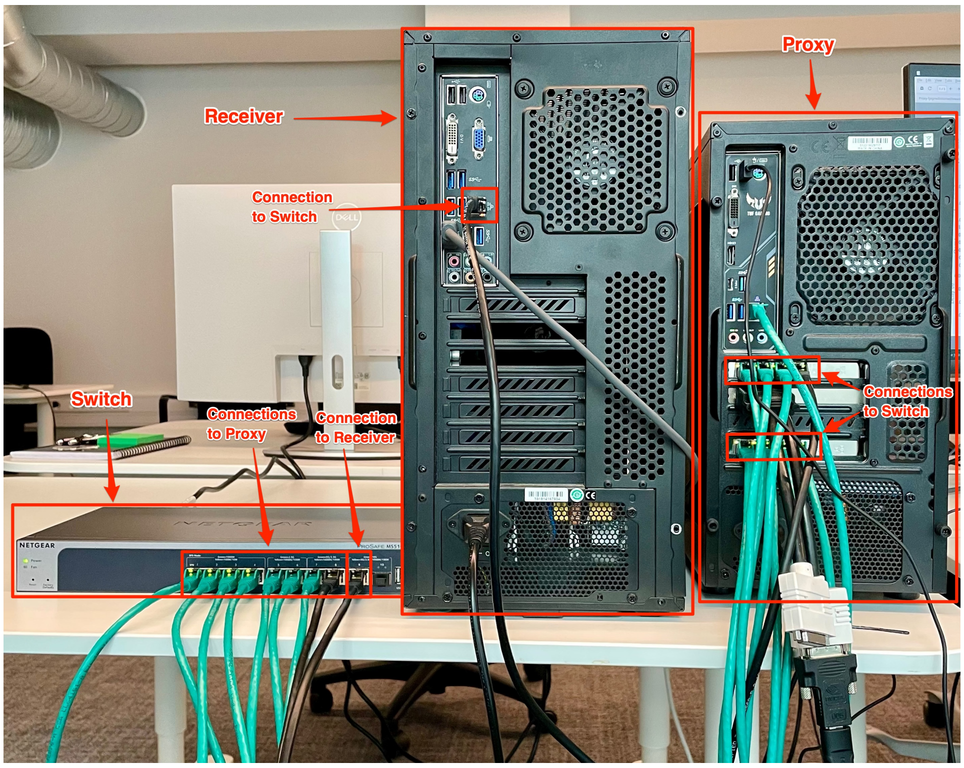


Figure 6.1: The setup of the hardware-based testbed. It consists of a machine that runs the proxy, another machine that runs the receiver, and a switch between them.

particularly challenging, as every component has to be able to deal with such speeds without slowing down the traffic. The machine that runs the proxy needs to run a multi-threaded program efficiently. The switch between the devices has to be able to forward packets from the proxy to the sender without packet drops, and the cables used must operate at gigabit speed. In the following section, each of the hardware components used in the testbed will be explained and why they were chosen for this testbed.

The Proxy machine

The machine that runs the proxy needs to be able to run as many threads as the proxy needs. As mentioned in chapter 5, the proxy runs numerous threads. The specifications of the machine used to run the proxy are shown in Table 6.1. It has a processor with six CPU cores and two threads per core. That means that it can run 12 threads concurrently, which is less than what is used by the proxy. Furthermore, the network cards used between the proxy and the switch are Intel X710-T4L. This is a network adapter that can operate with very high data rates, up to 10 Gbps [55]. Two such adapters are used in the proxy, with four interfaces available per adapter, enabling eight outgoing interfaces from the machine.

Table 6.1: The relevant specifications for the machine used to run the proxy.

Processor	AMD Ryzen 5 2600 Six-Core Processor (2 threads per core)
Kernel	5.17.1-3-MANJARO
Operating system	Manjaro over Arch Linux (64 bit)
Network cards	Intel X710-T4L
Memory	8 GB

The speed of the CPU will make an impact on how well the proxy will perform. A CPU that runs on a higher frequency will be able to execute more operations. Thus it can handle more load. The processor used in the machine that runs the proxy, see Table 6.1, has three different CPU frequencies that it can run at: 1.55 GHz, 2.8 GHz, and 3.4 GHz. Which frequency each of the 6 CPU cores uses depends on what the governor for the CPU chooses. Different types of governors can be used. There are governors that either select a frequency for the CPU statically (*performance*, *powersave* and *userspace*) or dynamically select it as the load changes (*ondemand*, *conservative* and *schedutil*). The default governor is *schedutil* which uses CPU utilization information provided by the scheduler to make its decisions. We decided to use the governor that is called *performance*, which is a static governor that selects the highest possible frequency for the CPU.

The Receiver machine

The receiver machine is only receiving packets from the proxy. Nothing is done with the content it receives, and no messages are sent back to the proxy to do flow control, i.e. tell the proxy to reduce its sending rate. The point of this machine is only to serve as an external machine that the proxy can send its data. Having a separate device is essential in this context because we want to set up multiple real connections between the proxy and the receiver to emulate a wireless mmWave network.

Switch

A switch is used to have multiple paths between the proxy and the receiver. The switch we use in this testbed is NetGears' 8-port multi-gigabit ethernet smart switch MS510TX [56]. This switch has eight ports for incoming packets and one uplink port for outgoing packets. This means we can have eight interfaces between the proxy and the switch and one interface between the switch and the receiver.

The switch support different data rates for the various interfaces it has. Four of the ports support up to 1 Gbps, two of them can handle data rates up to 2.5 Gbps, and the last two of the ports support traffic with speeds up to 5 Gbps. All of them support traffic down to 100 Mbps. For the uplink port, which is connected to the receiver machine, traffic up to

10 Gbps is supported. However, the receiver machine does not support traffic speeds up to 10 Gbps, it only supports traffic up to 1 Gbps. This means that the switch may need to drop packets it receives from the proxy if the proxy tries to send packets at a faster rate than 1 Gpbs. As the goal of the testbed is to test the proxy and how the different path managers perform in the proxy, we do not care that packets get dropped in the switch.

The MS510TX switch is considered to be an intelligent switch, where not only switching can be performed but also routing. This can be set up using the graphical user interface (GUI) that comes with the switch. This is not useful for our use case, but the GUI did make it easier to verify that everything on the switch was configured correctly.

Cables

To support traffic with the data rates we need to operate on, the choice of cables is important. This is because we do not want the wires to be the bottleneck. Two types of cables are used: two of type Cat6a and seven of type Cat6. Type Cat6a cables can support 10 Gbps network connections, and one of them is used between the switch and the receiver, and the other is used between one of the high-speed ports on the switch to the proxy. For the remaining seven connections between the switch and the proxy, cables of type Cat6 are used. They support traffic up to 1 Gbps.

6.2.3 Sending traffic to the proxy

The sender will send packets to the receiver via the proxy in a real system. However, in the testbed, the sender is not an external machine that sends packets but rather placed in the same machine that runs the proxy. We did not have access to three computers, thus the sender and the proxy had to be placed on the same machine.

Generating traffic to be sent to the proxy is not a trivial task when we want to achieve a sending rate of 2 Gbps (the sending rate used in the simulation experiments in chapter 4). Multiple iterations were needed to set up a sender that can send packets to the proxy at a gigabit rate. The first approach in setting up the sender was introducing a new thread that will send packets to the sender via localhost. This means that the proxy will be receiving packets via localhost, i.e using IP address 127.0.0.1.

To achieve a sending rate of 2 Gbps, the sender must send a packet every sixth microsecond. This means that the sender has to send a packet, then sleep for six microseconds before sending a new packet. The challenge is to be able to sleep for only six microseconds. As discussed in subsection 5.2.5 different ways to sleep exist in Julia. When using `sleep()` [42], where the minimum sleep time is one millisecond, a sending rate of 6 Mbps is achieved. This is too slow, as a mmWave network operates in hundreds of megabit or gigabit speeds.

Another function that can be used is `systemsleep()` [43], which yields a speed of 116 Mbps when used. This is much better than the other one but still very much less than the goal of 2 Gbps. Furthermore, using this function call increases the consumption of CPU power spent on running the kernel, from 20% (used when running `sleep()`) to 53% (this was found by running Listing 6.1 while running the proxy, and then taking the mean of the %system column). As discussed in subsection 5.2.7, using a lot of the CPUs to run kernel-level code will decrease the performance of the proxy since less CPU will be available to run it.

Listing 6.1: Used to find the procentage of CPU used to run the kernel for different sleep-functions. 38257 is the process ID for the Julia program running the code.

```
$ pidstat -p 38257 -u -I -l 1
```

Since `sleep()` and `systemsleep()` has too high granularity, which prevents the sender from sending packets with an interval of six microseconds, `nanosleep!()` can be used, which can sleep with nanoseconds precision. When switching out the sleep function used previously with `nanosleep!()`, which uses C's `clock_nanosleep`, a sending rate of 555 Mbps is achieved. The sending rate is dramatically increased by changing how the sender sleeps between sending the packets, but it is still not at the desired rate of 2 Gbps. This can be explained by looking at the CPU consumption again and how much the CPUs execute things at the kernel level. By running Listing 6.1 while running the proxy with the sender using `nanosleep!()`, 74% of the total CPU power is used to run kernel-level code. This will mean that less CPU power is available to run the proxy, thus it is not able to receive at a higher rate than 555 Mbps.

Another way of generating traffic is not to send over sockets at all, but to let the proxy generate the packets itself. Since the primary focus is on multipath and getting the proxy working in a multipath environment, having a sender that sends packets to the proxy via a socket is unnecessary. The important thing is that packets get placed in the queue at a specific rate. Removing the overhead of sending over localhost will also reduce a lot of the processing associated with it since we can avoid processing every packet twice (sending to localhost and receiving from localhost).

To let the proxy generate the traffic itself means that the same thing will be done as before, but rather to have a sender send a packet every sixth microsecond, the proxy will put a packet in the queue every sixth microsecond. As this is done in the proxy, `ClockedNanoSleep()` (see subsection 5.2.5) can be used. By running Listing 6.1 again, the total percentage of CPU power used went down to 26%, which is similar to the first approach where `sleep()` was used. This is very good, as it shows that doing it this way is less demanding on the CPU, thus giving more available CPU power to run the proxy.

However, it did not yield a sending rate similar to 2 Gbps. A rate of 1.3 Gbps was achieved. This is far better than the other approaches mentioned, but it is still not the same rate as used in the simulator. We could not achieve a higher rate because of the limits the hardware used for this testbed has. We chose to test the proxy with a sending rate of 1 Gbps to keep it simple. Since `TimeQueue` sleeps for 50 microseconds between each time it checks its ingoing queue, as described in subsection 5.2.5, it is not necessary to sleep less than 50 microseconds. This means that a packet is not placed in the queue every 12 microseconds (which is the interval needed to send at 1 Gbps), but 5 packets are placed in the queue every 60 microseconds. This will give an average sending rate of 1 Gbps. The proxy will not be tested with the same sending rate used in the simulator, however, 1 Gbps is still very high, so the proxy will still be tested in a challenging environment.

6.2.4 Communicating between the machines

For the proxy to be able to communicate with the receiver, IP addresses need to be set up. Each interface in the proxy and the interface in the receiver are assigned an IP address. This is done manually to have maximum control over the addresses and set it up correctly. The IP addresses are set by using the GUI interface on Manjaro, where one can see all the network interfaces one has and edit them. In Figure 6.2 the interface *enp6s0f0* is getting the IP address 192.168.0.1.

Routes need to be set up to ensure that the different interfaces are treated like different connections and not as one connection. This will lock a specific IP address to a specific interface, which will ensure that sending packets from different IP addresses also means sending them over the different interfaces. This was done by first creating a route (see Listing 6.2) and then adding it as a rule (see Listing 6.3) for all of the eight interfaces.

Listing 6.2: Adding a IP route

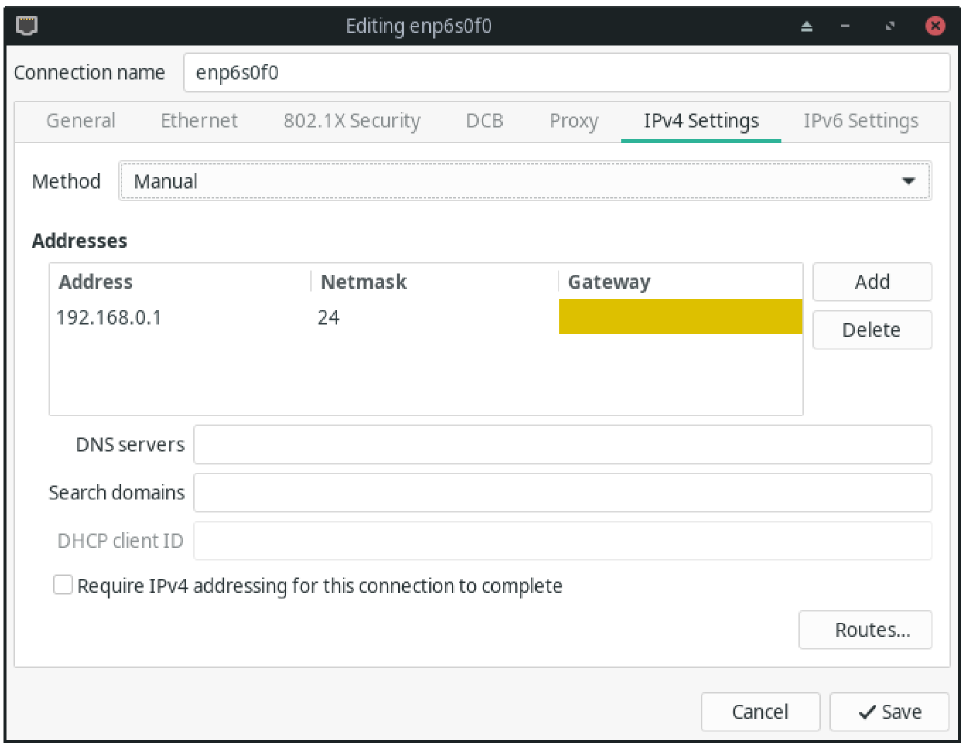
```
$ sudo ip route add default via 0.0.0.0 dev enp6s0f0 tab 1
```

Listing 6.3: Adding a IP rule

```
$ sudo ip rule add from 192.168.0.1/32 tab 1 priority 500
```

6.2.5 Changing the capacity of the interfaces

To emulate a mmWave network the same way as was done in the simulator in chapter 4, changing the capacities of the different paths is essential. Since mmWave is a wireless technology that is very sensitive



Connection name: enp6s0f0

Method: Manual

Addresses

Address	Netmask	Gateway
192.168.0.1	24	

DNS servers:

Search domains:

DHCP client ID:

☐ Require IPv4 addressing for this connection to complete

Figure 6.2: Setting up the IP addresses to be used in the proxy and the receiver.

to blockages, the capacity of the connection will also vary a lot. This we wanted to emulate in the testbed by periodically changing the capacity of the paths.

To change the capacity for a given interface is not as trivial as we initially thought. First, we tried using *ethtool*, which can be used to query and control network and hardware devices. Listing 6.4 shows how it can be used to change the speed on interface *enp6s0f0* to 100 Mbps. However, when we tried to do this, we got an error telling us that we did not have permission to change it.

Listing 6.4: Chaning the capacity by using Ethtool

```
$ sudo ethtool -s enp0s3 speed 1000
```

Next, we tried changing the capacities of the interfaces between the proxy and the switch using the switch' GUI. For each switch's interface, the speed can be manually set. However, it can only be set to a few predefined values. We also found out that setting a link to have a speed of 100 Mbps was not possible, even though it should be able to run at that speed. Furthermore, manually setting the speed for each interface is not a viable solution. A solution to this could be to use Simple Network Management Protocol (SNMP). SNMP enables sending simple commands to the switch. However, changing the speed is not one of the supported commands that can be sent over SNMP to this particular switch.

The last attempt to change the speed was to use a traffic controller [57] that can change the queuing discipline of an interface. We chose to set every interface to the MQPRIO qdisc [58], which is a queuing discipline that maps traffic flows to a hardware queue, where several options can be set, where the most important option is the *max_rate* option. With this, we can set the maximum bandwidth rate limit for a particular interface.

With Listing 6.5 we can assign interface *enp6s0f0* a MQPRIO qdisc, and with Listing 6.6 we can change the speed to be 100 Mbps.

Listing 6.5: Insert a qdisc of type mqprio

```
$ sudo tc replace dev enp6s0f0 root mqprio num_tc 1 hw 1 mode channel
  shaper bw_rlimit min_rate 0Mbit max_rate 1Gbit map 0 0 0 0 0 0 0 0
  queues 1@0
```

Listing 6.6: Changing the speed to 100 Mbps

```
$ sudo tc change dev enp6s0f0 root mqprio num_tc 1 hw 1 mode channel
  shaper bw_rlimit min_rate 0Mbit max_rate 100Mbit map 0 0 0 0 0 0 0 0
  queues 1@0
```

However, a bug in the Intel driver made this change impossible. Intel was notified about this bug, and they are working on resolving it, but the fix is not released in time for it to be used in this thesis.

Furthermore, we tried to set up two traffic classes for each interface so that each traffic class could be assigned different data rates. Then, packets sent from the proxy would be given different priorities based on the path's availability. However, another bug in the Intel driver prevented us from assigning multiple traffic classes to one interface.

After testing out various ways to change the capacity of the eight interfaces between the proxy and the switch, where none of them worked, we ended up using four paths, where each path is gives two interfaces. By using Listing 6.5 to set the speed of an interface, we could set up one LoS speed and one NLoS speed for four paths. Whenever a paths switch from LoS to NLoS, or vice versa, the interface used to send packets will also change. This enables us to test the proxy under a more realistic scenario, than if all the path's capacites were static, though with fewer paths than we wanted.

Chapter 7

Testing the proxy implementation on an emulated mmWave network

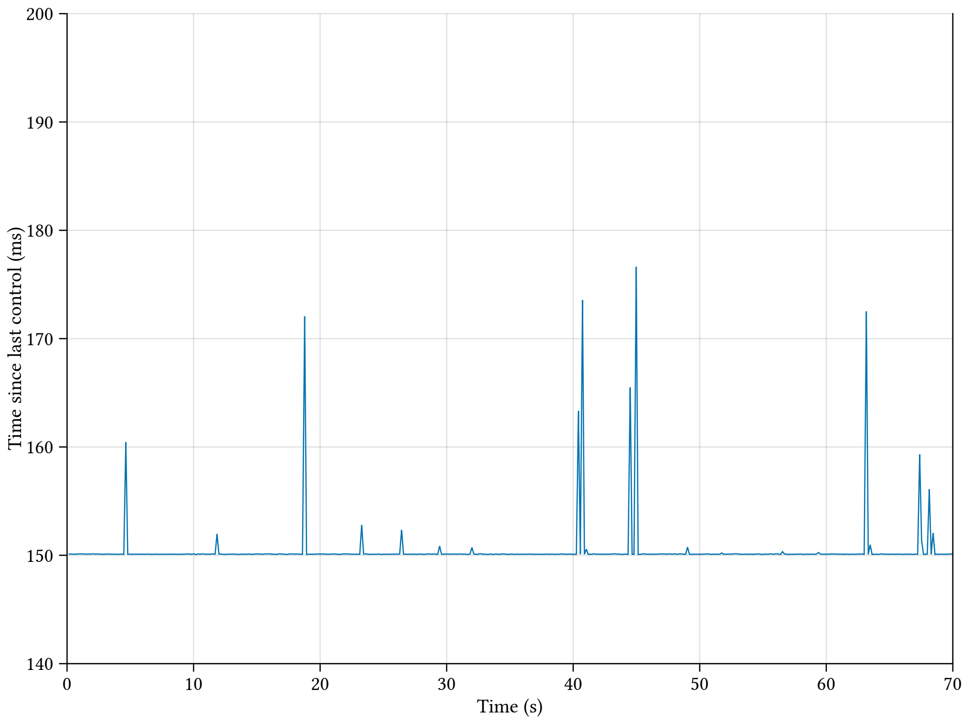
In this chapter, the results from testing the performance of the proxy and testing how the different path management models affect the performance of the proxy are presented. How accurate the timing is in the proxy will heavily affect how often the predictive control is triggered, thus this will be tested. Next, how long it takes to do the predictions once a predictive control is triggered will decide if a prediction-based path management scheme is a feasible solution or not. It is also important to investigate how the proxy utilizes the CPUs available on the testbed machine, as it can indicate how efficiently the proxy has been implemented. Lastly, how the different path management models presented in chapter 3 affect the performance of the proxy is presented.

This section will answer the final research question defined for this thesis: *How do different path managers perform in a proxy that operates in an emulated environment?*.

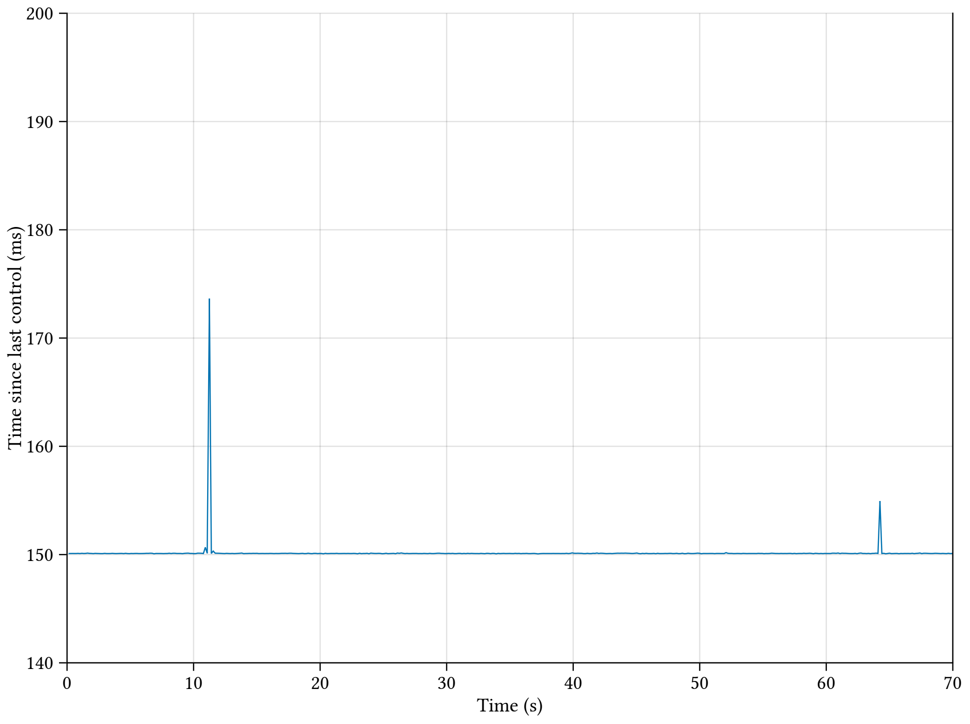
7.1 Time between each predictive control

A path management model that uses a reactive approach in managing the different paths will trigger a control every time the queue surpasses a given threshold. This means that changes in the queue will trigger a reactive control so that a path can either be added or removed. This is not the case when using a predictive approach since the goal is to act before the queue gets the chance to surpass the threshold. Therefore, the predictive control is triggered periodically by the proxy itself, regardless of how the queue evolves. The goal is to trigger the predictive control every 150 ms.

To trigger something periodically in a real system can be challenging because no timer can guarantee that it will trigger something precisely at a given time. It will probably never do that. However, the goal is to reduce this overhead as much as possible. As described in



(a) CDF-based prediction



(b) FPT-based prediction

Figure 7.1: The time between each control interval during one run of the proxy using the purely predictive based control models.

subsection 5.2.6, different ways of scheduling the predictive control were tested out. The most important thing is that the time between each predictive control is shorter than the time horizon, i.e. the amount of time in the future that the prediction is trying to predict the queue. If the predictive control is triggered more rarely, there will be gaps in between where the path manager has no prediction of the queue, hence it has no clue of what is happening with the queue. This will yield low performance as the queue may grow without the path manager being able to add a path.

The proxy uses a while-loop with the sleep function `ClockedNanoSleep()` to periodically trigger the predictive control (see subsection 5.2.6). Figure 7.1 shows the time interval between each time a predictive control is triggered. Figure 7.1a shows the results when using purely CDF-based prediction and Figure 7.1b shows the results when using purely FPT-based prediction.

The results show that the predictive control is not triggered any earlier than 150 ms after the last predictive control. Furthermore, for the most part, it is triggered no later than 150 ms either. This is good because it demonstrates that utilizing `TimeQueue` delivers millisecond-level accuracy. There are occasional spikes in both figures, where the duration between two predictive controls is higher than 150 ms. As this is a real system, the proxy will be working on multiple things simultaneously, where the load will switch between the CPU cores. This can affect the timing because a thread that is waiting for a timer to expire can be blocked so that it takes longer before it can be executed, i.e. the predictive control is triggered later. However, the predictive controls are not triggered any later than 200 ms, and there are few spikes.

7.2 How long do the predictions take?

The path manager triggers the predictive control periodically. Each time the predictive control is triggered, it must predict how the queue will look for the next time horizon. This prediction includes running calculations using multiple threads. Using multiple threads will decrease the overall time it takes to finish the calculations, i.e. a result will come faster. If it takes too long to run the calculations, using a predictive approach for control purposes will not be the best approach.

How long the calculations take is tested by running the calculation 1000 times, with different numbers of paths. This is done with both the CDF-based and the FPT-based prediction. Table 7.1 is the result of these tests.

As seen in Table 7.1, the more paths used, the longer the calculations take. This is reasonable since the Markov model that needs to be solved will have more states as the number of paths increases, thus the matrix to be solved will increase. The full state model gets solved when there are 1 or 2 paths, else the simple weighted model gets solved (the models

Table 7.1: Execution time of the prediction showing the 90% percentile results after 1000 runs. When there are 1 or 2 paths, the FULL model is used, when there are 3 or 4 paths, the weighted model, WQx, is used. 4 threads are used for the calculations.

Number of paths	Prediction	
	CDF-based	FPT-based
1	2.3 ms	1.8 ms
2	5.2 ms	4.2 ms
3	5.3 ms	4.4 ms
4	9.0 ms	7.6 ms

are explained in section 3.2). The result from this is similar to the fixed costs that were used in the simulator in chapter 4. They are within reasonable time for what can be spent on control. However, suppose the number of paths gets higher, the time spent on solving the model will also increase. Therefore, it is important to be mindful of this so that the path management does not spend all its time solving the calculations, thereby losing the advantage of a predictive approach.

7.3 Proxy's CPU utilization

How much CPU the proxy utilizes indicates how efficient the code is. If the proxy uses a lot of the CPU, the proxy is probably doing something wrong. Using too much CPU can also lead to the proxy freezing. Keeping the CPU load as minimal as possible is therefore preferable.

Figure 7.2 is the results of running the proxy with two different path management schemes (CDF-based prediction and FPT-based prediction) and sampling the CPU load every second by using Listing 7.1. The proxy will put packets in the queue, send packets over the different paths, do path management, activate and deactivate paths, and switch the LoS dynamic of each path simultaneously, as described in chapter 5.

Listing 7.1: Find CPU load of proxy by sampling every second. 38257 is the process ID for the Julia program running the proxy.

```
$ pidstat -p 38257 -u -I -l 1
```

The results show that the proxy uses around 50% of the total CPU capacity available on the testbed machine. This tells us that the proxy does not overuse the CPU, thus we can be more confident about the proxy's implementation. However, the proxy implemented in this thesis is not deployable as is. For example, the proxy supports only unidirectional flows and only one user. A real-world proxy has to support traffic going in both directions, and it probably has to support multiple users. This will increase the CPU load. However, the hardware that

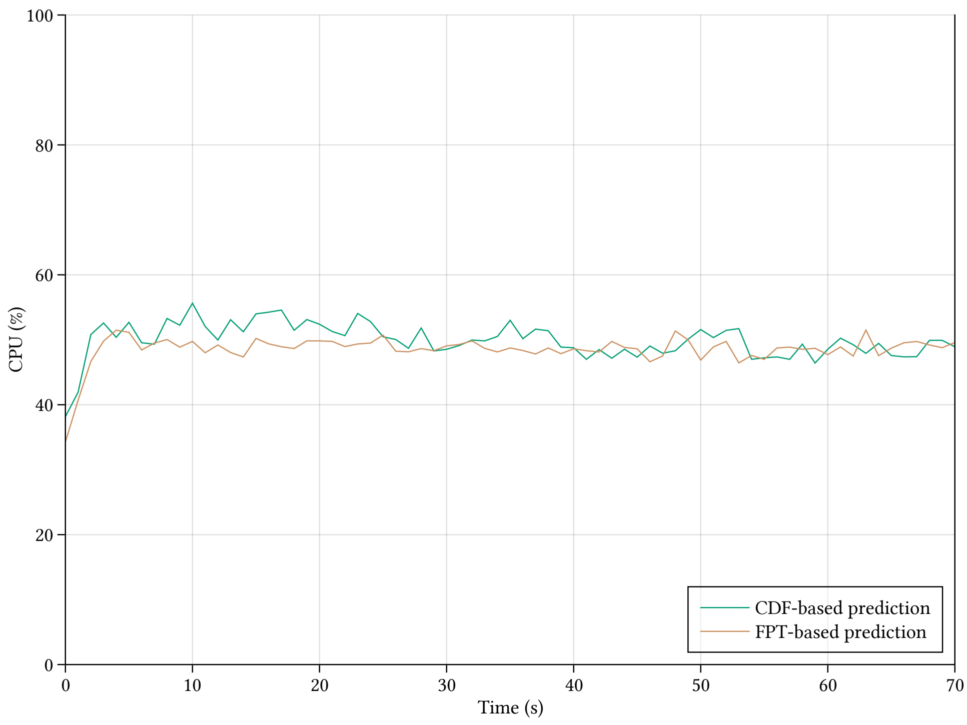


Figure 7.2: The CPU consumption of running the proxy with two different path management models: CDF-based prediction and FPT-based prediction.

will be used to run a proxy in a real-world deployment will be a more powerful machine that can handle the proxy's load.

7.4 Testing the proxy using different models

The main goal of implementing a proxy to be used in a mmWave network is to test the different path management schemes introduced in chapter 3, thus this will be tested next. The experiment setup (see Table 7.2) is similar to the setup used to test the different models in the simulator. Packets will be put into the queue at a rate of 1 Gbps. A total of four mmWave paths are available, where each path has an LoS capacity of 900 Mbps and an NLoS capacity of 100 Mbps. Each path will on average be in LoS for 3 seconds and in NLoS for 2.5 seconds. The path change delay, i.e. the time it takes for the operator to add/remove a path, is set to 20 ms. The high queue threshold, Q_HT, is set to 500 packets, and the low queue threshold, Q_LT, is set to 250 packets for the CDF-based models and 1 packet for the FPT-based models and the reactive model.

Figure 7.3 shows the results of running the experiment with the five different path management models:

- Reactive control (Figure 7.3a)

Table 7.2: Emulation parameters

Sending rate	1 Gbps
Number of paths	4
LoS capacity	900 Mbps
NLoS capacity	100 Mbps
Average time in LoS	3 s
Average time in NLoS	2.5 s
Path change delay	20 ms
Q_HT	500 packets
Q_LT	250 or 1 packet [1]
Packet size	1500 B
Predictive control interval (*PC_min_T*)	150 ms
Reactive control interval (T_R)	50 ms

[1] 250 when CDF-based prediction is used, otherwise 1.

- CDF-based prediction (Figure 7.3b)
- CDF-based prediction with reactive control (Figure 7.3c)
- FPT-based prediction (Figure 7.3d)
- FPT-based prediction with reactive control (Figure 7.3e)

For each experiment, four different metrics are collected: the number of paths being used, the available capacity of the used paths, the queue size (show a band of max and min queue values achieved over a timespan of 200 ms), and the packet delay within the proxy (show a band of max and min delay values achieved over a timespan of 200 ms). The packet delay is the time each packet spends in the proxy, i.e. delay caused by queuing. Packets drops are marked as black dots in the *Capacity* plot and the *Delay* plot. Whenever a reactive control is triggered when CDF-based prediction with reactive control or FPT-based prediction with reactive control is used, this is marked as a red dot in the *Paths used* plot.

As the sending rate is 1 Gbps, it is expected that the queue will increase as the total available capacity is below 1 Gbps. In Figure 7.3a where a reactive control model is used, spikes in the queue appear as the total available capacity drops below 1 Gbps. When this happens, the proxy will try to add a path. If there are no more paths to add, the queue will start to fill up with packets, and the proxy will eventually need to drop packets when the queue is full. This happens twice in Figure 7.3a: after 10 seconds and 25 seconds.

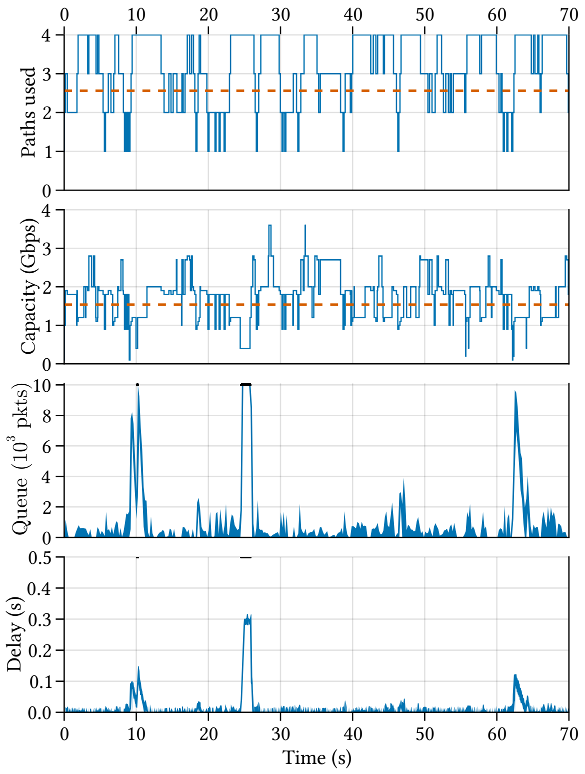
The reactive control model struggles to keep the queue consistently low. This is because it will remove a path when the queue only has 1 packet and add a path when the queue has more than 500 packets. So when one path is used, the queue will increase since one path in LoS has a lower capacity than the sending rate. When the queue has more than 500 packets, a path will be added so that the total available capacity is

1.8 Gbps (if both paths are in LoS). However, since a path change delay of 20 ms is added, a path will not be added right away, so the queue will continue to grow till a path gets added. When a path is activated, the queue will drain, and eventually, when the queue is empty, a path will be removed. When this happens repeatedly, the queue will have spikes. This happens multiple times in Figure 7.3a, where the number of paths switches between one and two, consequently creating spikes in the queue.

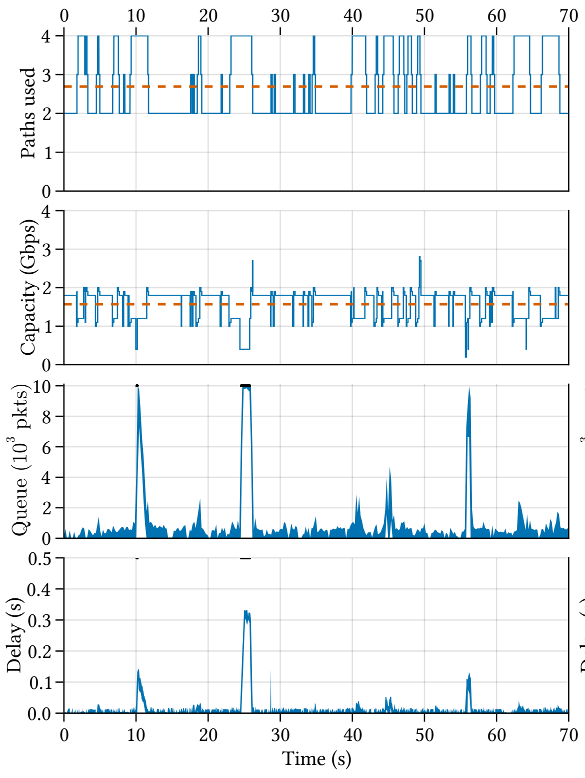
A predictive-based model will mitigate this by not removing a path if the likelihood of the queue increasing when a path is removed is high. This is seen in Figure 7.3b, Figure 7.3c, Figure 7.3d and Figure 7.3e, where the number of paths is never less than two. This has an effect on the queue, in that the queue does not have the spikes seen when using a reactive control model, but is on a consistent level.

That the queue is seemingly not empty when the sending rate is lower than the total available capacity can have numerous explanations. First of all, packets are put in the queue and taken out of the queue in batches, and not at a constant rate (subsection 6.2.3 explains how packets are put in the queue and section 5.2.7 explains how packets are taken out from the queue). This can lead to the queue having more bursty behavior, which can be the reason why the queue level is more than expected.

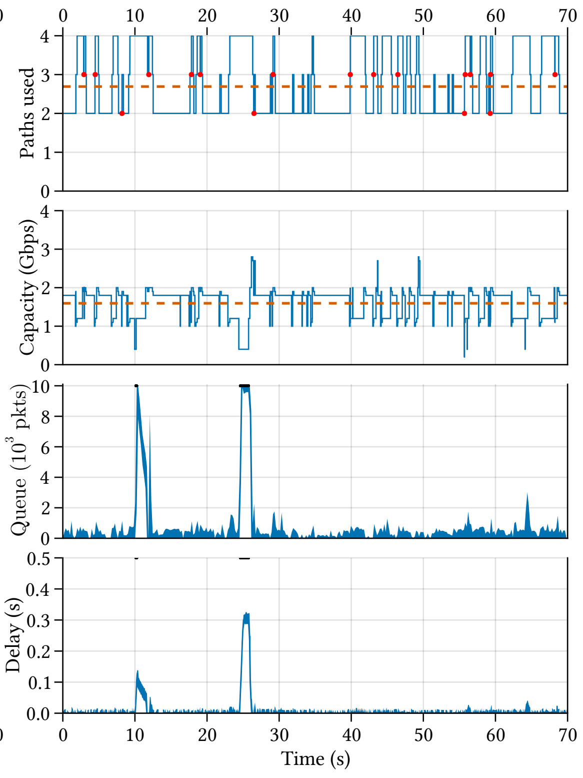
Another reason can be how packets are being scheduled on the



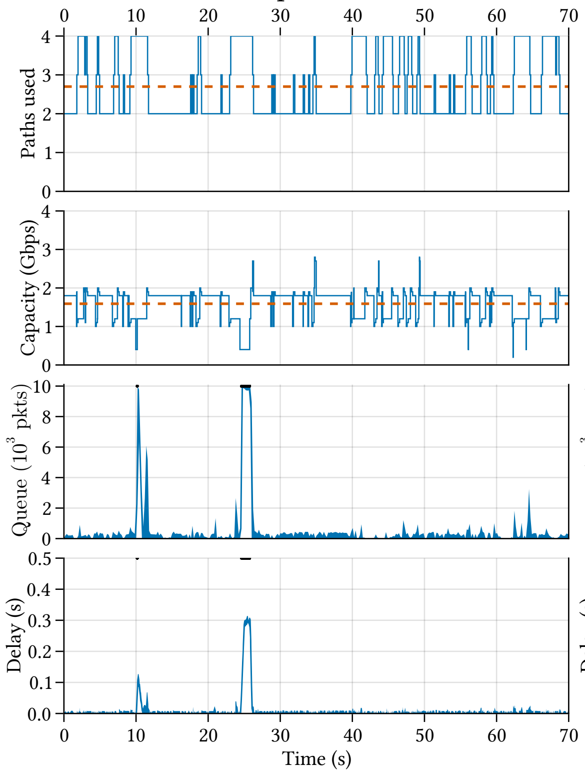
(a) Reactive control



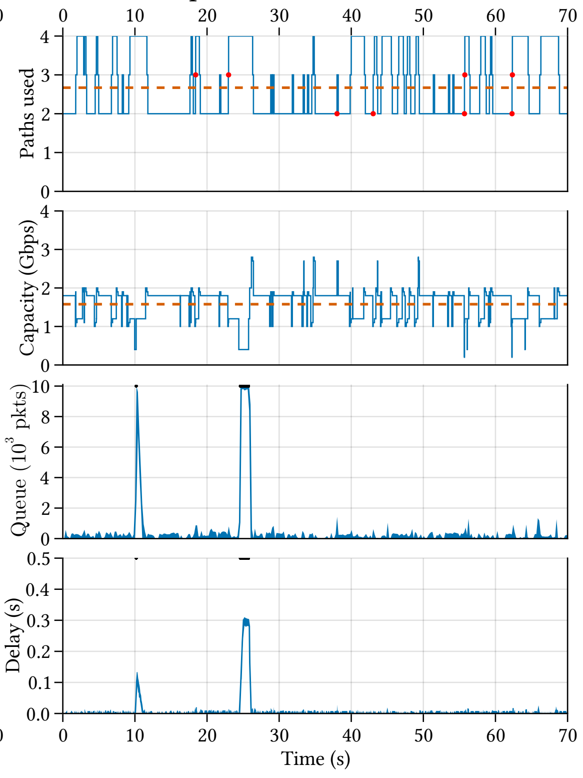
(b) CDF-based predictive control



(c) CDF-based prediction with reactive control



(d) FPT-based predictive control



(e) FPT-based prediction with reactive control

Figure 7.3: Running the proxy with different path management schemes.

various paths. As explained in subsection 5.2.7, a path greedily takes packets out from the queue to send them, without regard for their capacity, hence, many packets can be transmitted over low-speed paths. This will slow down the proxy, in that packets in the queue will need to wait longer before they are taken out and transmitted.

A third reason for the high queue levels can be of how timing is done in the proxy (see in subsection 5.2.5). As the sending threads sleeps when they are not taking packets out of the queue, as explained in subsection 5.2.7, how accurate the timing is will affect how efficiently packets are taken out of the queue. A more accurate sleep will give a shorter sleep time, which enables sending packets over that path quicker, which can prevent the queue from growing as much.

The accuracy of `TimeQueue` depends on two things: how accurate `nanosleep!()` is and how accurate Julia's `Channel` is. `nanosleep!()` was found to be very accurate when testing different ways to do timing in the proxy (see subsection 5.2.5). Julia's `Channel` on the other hand, was found to be a bit slow when elements were put in and out of it at a fast rate (see subsection 5.2.3). Elements are not necessarily put in and out at a fast rate in this use case (unless a task call `ClockedNanoSleep()` repeatedly after each other with a short sleep time). A call to `ClockedNanoSleep()` will block because the waiting task is calling `take!()`, effectively blocking until `TimeQueue` put something in the channel by calling `put!()`. The task will not finish its sleep before it sees an element in the channel and can take it out. How soon after something is put in the channel before the waiting task can take it out, depends on how soon that task is scheduled in an available thread, i.e. it depends on how Julia 1.7 schedule its tasks to its threads. As explained in subsection 5.2.5, Julia 1.7 do not have the most efficient way of scheduling tasks to the various threads, which can lead to a call to `take!()` use a longer timer to take something out of the queue than it should have.

The reason for the seemingly high queue levels is most probably a combination of all the above. It is also important to note that the queue and the delay graphs are a band of max and min values achieved over a timespan of 200 ms. This means that the queue can be empty most of the time, without showing it. If the queue has a small spike, then the max value will be updated, which will be reflected in the plot. Furthermore, the emulator runs on a real system, thus some noise has to be expected since the machine has to maintain its own system while running the proxy. And since packets are going through the proxy at a very high rate, where microsecond precision is required, things happening on the machine can affect the proxy.

Packet drops will be inevitable when all of the paths are in NLoS simultaneously. As there are only four paths used in these experiments, the possibility of all of them being in NLoS is greater than if eight paths were used (as in the simulator). It is seen that the proxy is able to drain the queue after a period of continuous packet loss. This is very good, as it shows that the proxy is able to send packets quicker than it receives them.

Figure 7.3c and Figure 7.3e shows the results when using CDF-based prediction with reactive control and FPT-based prediction with reactive control, respectively. The reactive control is triggered 17 times in Figure 7.3c and 8 times in Figure 7.3e. Compared to how often the predictive control is triggered in the purely predictive-based models – around 470 times – this is good. It shows that the reactive control does not have to be triggered that often since the predictions of the queue are generally correct.

When comparing Figure 7.3b with Figure 7.3c and Figure 7.3d with Figure 7.3e, it is seen that spikes in the purely predictive-based figures are lessened when a reactive control is added. This shows that a predictive-based path manager with reactive control is better at coping with unforeseen spikes in the queue than a purely predictive-based path manager.

7.4.1 CDF of the delay

The primary goal of the proxy is to provide reliable consistent communication, i.e. a steady transmission rate with low delay, hence looking at the delay is important. Plotting a CDF of the delay will better illustrate the difference between the delays found in Figure 7.3, hence this is done in Figure 7.4. From this figure, we can see that the reactive control has the highest delay of them all. Furthermore, using FPT-based prediction gives a lower delay than using CDF-based prediction. When FPT-based prediction is used, an action will be taken if the likelihood of the queue surpassing a threshold once is high. This means that a path might get added sooner, compared to when a CDF-based prediction is used where the whole queue distribution must be over/under a certain threshold for a path to be added/removed. Figure 7.4 also shows a small performance boost, in terms of smaller delay, when adding a reactive control to a predictive-based path manager.

7.4.2 Comparing simulations results with emulation results

Comparing the simulation results in section 4.3 with the results from the emulation experiments has to be done carefully. The simulator has no noise since everything is controlled, so comparing the different path management schemes with each other can be done, since running an experiment with the same parameters will give the same results. This is also why the first evaluation of the path manager was done in a simulator. Repeatable experiments are easier to compare. When evaluating the proxy in the emulator, things are a little different. Emulation experiments are only approximately repeatable. Since emulations aim at controlling some things while keeping other things out of the programmer's control, a more realistic scenario can be simulated. However, it also means that two experiments in an emulator

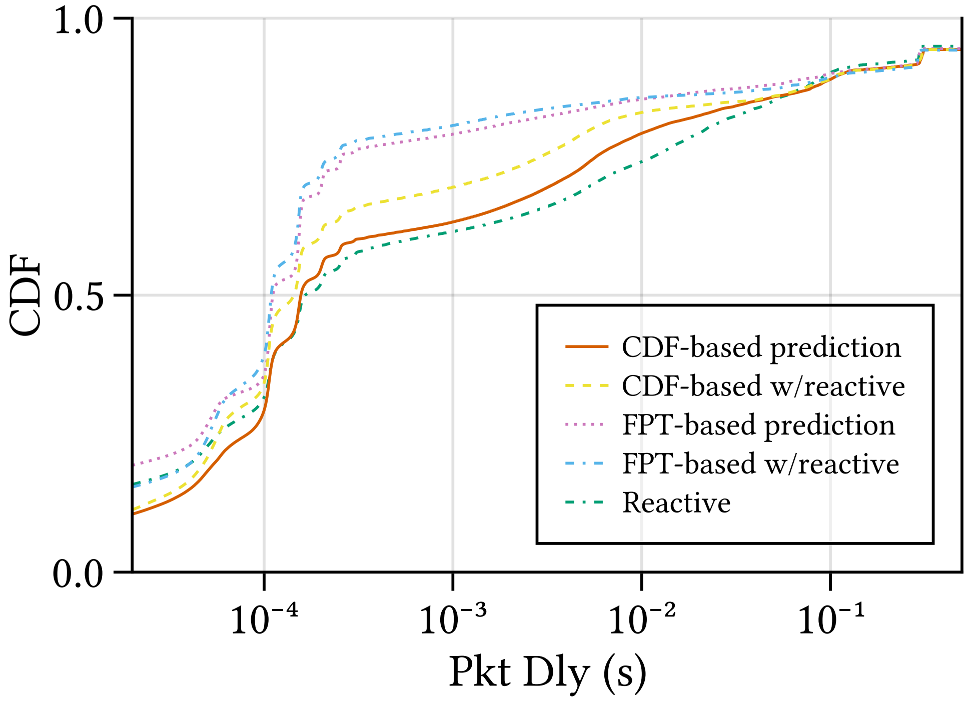


Figure 7.4: Delay CDF from the same experiment as Figure 7.3, where different path management schemes are used.

will never be equal, hence, when comparing two emulation results one needs to be mindful of this.

Even though comparing the simulation results in section 4.3 with the emulation results in section 7.4 are more challenging, does not mean that it cannot be done, it just has to be done carefully. The simulation results show that a predictive-based path manager does perform better than a reactive control. Furthermore, adding a reactive control to a predictive-based control performs even better, as unforeseen queue spikes can be dealt with quicker. This is also seen in the emulation results. The difference between a reactive control and a predictive-based control is not as distinct as they were in the simulator but can be seen in Figure 7.4.

Figure 4.1 is the equivalent of Figure 7.3, where the first is the results from running the experiments in the simulator and the second one is the results from running the experiments in the emulator. Both are single-run experiments. They are not exactly the same experiment. In the simulator, eight paths and a target rate of 2 Gbps were used. In the emulator, only four paths were used and only the LoS/NLoS dynamics were captured (not the shadowing fading effect). The effect of this is seen in the *Capacity* plots in the figures, where the capacity fluctuates much more in Figure 4.1 than it does in Figure 7.3. Furthermore, a sending rate of 2 Gbps was not achieved in the testbed, as the proxy only received at a rate of 1 Gbps.

A noticeable difference between Figure 4.1 (simulation results) and

Figure 7.3 (emulation results) is the delay, where running the proxy in the emulator gives a higher delay than running the experiments in the simulator. Note that the scale of the delay plots is different in the two figures. In Figure 4.1 a log-scale is used, while Figure 7.3 uses a linear scale to show the delay. Nevertheless, running the emulations gives higher delay than running the simulations. This is also seen when comparing Figure 4.2 with Figure 7.4 (the first is the results from running the simulation 100 times and plotting the delay CDF and the second is the delay CDF of a single-run emulation experiment), where the emulation results have a much higher delay, especially for the predictive-based models than what is seen in the simulation results. Given that the simulator has a higher sending rate than the emulator, this might be surprising. However, as the emulator is a real system, everything takes some time. This is not something that is considered in the simulator. Furthermore, as explained in subsection 5.2.5, the proxy has some problems with how multi-threading is handled in Julia 1.7, which affects the timing. This means taking packets out of the queue can be slowed down, i.e. the delay goes up. This also explains why the queue occupancy is generally higher in the emulated experiments than in the simulated ones.

7.5 Summary

Even though comparing the simulation results with the emulation results is challenging, they do show the same tendencies, i.e. a reactive control performs worse than a predictive-based control, and adding a reactive control to a predictive-based path manager does give some performance boost. However, the results from this emulation study are based on single-run experiments. For future work, a more extensive emulation study should be done, where the implementation could be tested under more realistic mmWave conditions, once this functionality is available in the testbed.

Chapter 8

Conclusion

This thesis investigates how different path management models affect the performance of a multipath mmWave proxy.

To mitigate the challenges a mmWave network faces (high fluctuations in the achieved capacity due to the mmWave link's sensitivity to blockages) a multipath proxy can be deployed in the network. The proxy splits the end-to-end connection at the Internet edge so that one connection is used between the server and the proxy, and multiple mmWave connections are used between the proxy and the client.

When introducing a proxy that supports multi-connectivity, multiple functionalities have to be implemented, with path management being one of them. The path manager is responsible for establishing new paths and tearing down unavailable or unneeded paths. How the path manager decides which paths should be operative and not can heavily affect the performance of the proxy.

This thesis proposes two new path manager schemes that combine a predictive and a reactive approach. The primary method is to use a predictive control, which predicts how the queue will look periodically. If the queue starts to fill up between these intervals, a reactive control can be triggered to add a path immediately instead of waiting until the next predictive control is triggered. This will ensure that a path will be added sooner if the queue quickly starts to fill up, thus the queue will drain more quickly. To evaluate the performance gains of using the proposed schemes, a comparative simulation study was done first, then the schemes were tested in an emulated environment.

A proxy has been developed to test how different path management schemes affect the performance of the proxy when running it in a simple testbed that emulates a mmWave network. The proxy has to be efficient and fast, as it has to deal with high data rates going in and out. Furthermore, it has to do multiple tasks simultaneously, thus multi-threading is an important aspect of the proxy. The proxy does suffer from Julia 1.7's inefficient thread handling, which has been attempted to minimize by reducing the number of blocking C-calls. However, since multi-threading is an essential part of the proxy's design, it will be affected by it.

The results from testing the various path management schemes show that using a reactive control model yields the lowest performance, in terms of a higher queue and higher delay. The reason for this is that a reactive approach is too slow, as the queue has time to fill up before a path gets added. Using a predictive-based control mitigates this by predicting that the queue will rise if another path does not get added, so it will add a path before the queue has a chance to fill up. This will give an overall lower queue, and lower delay. Furthermore, a small improvement in the performance is seen when comparing the results of using CDF-based prediction with reactive control and pure CDF-based prediction and comparing the results of using FPT-based prediction with reactive control and pure FPT-based prediction. It is expected that only a small improvement is seen when adding a reactive control to a predictive-based model since the purely predictive-based model should be able to predict the queue accurately most of the time.

Despite the limitations of the proxy, the emulation experiments show that using a predictive approach is better than a reactive one. Furthermore, as this is only a proof-of-concept study, continuing the work with the proxy, and overcoming the limitations of the testbed, can further strengthen the results.

To wrap up this thesis, I will summarise the answers this thesis provides to the research questions defined in section 1.2:

RQ1: **How can a path manager with both a reactive and a predictive control be designed?**
The path management schemes proposed in this thesis combine a predictive and a reactive control by using the predictive control as the main way of changing the number of paths, as explained in section 3.3. The reactive control is only triggered in the most critical scenario, which is when there are not enough paths so that the queue will increase. A limit, called the reactive control interval, prevents the reactive control to be triggered too soon after another path change, as it can be costly for the operator to change the number of paths too often.

RQ2: **How can a proxy be implemented to work in a mmWave network?**
When implementing the proxy, multiple things are important to consider to get a high-end proxy that aims to operate in a demanding network with mmWave links that have intermittent availability. One key element is multi-threading, which will allow multiple things to run concurrently in the proxy. Various modules exist in the proxy (see section 5.1), where each module runs simultaneously. The three main modules are packet handling, which puts incoming packets in the queue, the path manager responsible for the paths, and the packet scheduler, which takes packets out from the queue and sends them over an available path.

RQ3: **Is the Julia programming language an adequate choice for implementing a real-world proxy?**
Yes, Julia can be used to implement a real-world proxy, as explained in section 5.2 and tested in chapter 7. However, using Julia 1.7 to implement the proxy did impose some challenges, mainly due to the premature thread handling in Julia 1.7. Reducing the number of blocking C-calls (see subsection 5.2.5), and reducing the amount of kernel-level executions by sending packets more efficiently (see section 5.2.7) did reduce the overhead of thread scheduling sufficiently, enabling a fast-running proxy that can work in a gigabit-speed network.

RQ4: **How do different path managers perform in a proxy that operates in an emulated environment?**
Testing the proxy using different path manager models (see section 7.4) show that a reactive approach is not able to add a path quickly enough, thus the queue fills up. When using a predictive approach, the number of paths never drops below two, consequently keeping the queue levels lower. This is because it predicts that using one path will make the queue grow, thus it will not remove a path if there are only two operational paths. Furthermore, Figure 7.4 show a small performance boost of adding a reactive control to a predictive-based model. However, the results are based on single-run experiments, so a more extensive emulation study should be done to strengthen the results.

8.1 Future work

This section will first go through the proxy's limitations, which should be addressed in future work. Moreover, ideas on studying other things than what has been studied in this thesis are discussed.

8.1.1 Multi-threading in Julia 1.7

To enable the proxy to do all of the different tasks, multi-threading is used. However, as discussed in subsection 5.2.5, Julia 1.7 does not handle multi-threading in the best way. It statically assigns a thread to a task without regard to the load. This means that a task that executes a bad blocking call, like calling a blocking C-function, on a thread, will consequently block all execution on that thread. Suppose another task is assigned the same thread, then both will be blocked. This can lead to system freezing.

A new version of Julia, Julia 1.8 beta, uses a dynamic scheduling policy, which may resolve this issue [59]. Dynamically scheduling tasks to available working threads, instead of iteratively selecting threads, will prevent that busy threads are selected. This can help the issues regarding multi-threading found when developing the proxy for this thesis, as calls to blocking C-functions did make a static scheduler insufficient.

8.1.2 Packet scheduler

The scope of this thesis was to look into path management. However, in a multipath protocol, a packet scheduler is equally important. The packet scheduler is responsible for scheduling the different packets to the different paths. Which path that a packet should be sent over is an important decision because it can affect the application's performance negatively if done inefficiently.

Head-of-Line (HoL) blocking is when packets get delivered out of order at the receiver and the receiver requires in-order-delivery so it has to store packets while waiting for the first packet to arrive. This can be a big problem in a mmWave network where multiple mmWave connections are used. As the capacity of the different links may vary greatly (because of the LoS/NLoS dynamics of the different links), sending a packet over a low-capacity link will be delivered to the receiver much later than packets sent over a high-capacity link. This is something a packet scheduler can try to minimize, by selecting paths that are in LoS more often than selecting paths that are in NLoS.

A packet scheduler can also try to minimize the overhead of needing to resend packets due to packets being lost by using forward error correction (FEC). FEC is a technique where the proxy will send packets and repair packets so that the receiver can use the repair packets to recover lost packets [2].

Moreover, the packet scheduler can receive signals from the network that can be used when it decides where to send a packet. For instance, information about congestion in the network can be used by the proxy to prevent sending over certain links.

8.1.3 Reacting to other things in the network

Adding a reactive control interval to a prediction-based system is one way of compensating for unforeseen changes. However, there are other things that the proxy can react to. For instance, there may be a way for the network to signal the proxy that a link is going from LoS to NLoS. If the proxy knows that a path will be in NLoS in a short time, a new path can be added to prevent the drop in the capacity.

8.1.4 Different ways to deploy a multipath proxy

The proxy developed in this thesis is built to work in this particular setup, where there is a sender that sends packets, and a proxy that receives them and forwards them to the receiver using multiple mmWave links. However, this is not how a real system works. First, a real system will accept packets from both the sender and the receiver. Bi-directional transmission is not supported in the proxy developed in this thesis, as only packets from the sender are accepted. In the process of developing a deployable proxy, this must be included.

Furthermore, the setup used in this thesis assumes that the proxy has access to the outgoing queue of the various mmWave links. This is not given, and it depends on where the proxy will be deployed in the network and where the mmWave base stations will be deployed. If the proxy is connected to the different mmWave base stations by an ethernet cable, as in the testbed in this project, then this proxy would be similar to a proxy in a real system. However, if the different mmWave base stations are not placed in the same place that the proxy is placed, the proxy would not have direct access to their queue, consequently making the proxy developed in this thesis undeployable without alterations.

8.1.5 Decoupling the proxy and the path manager

When developing the proxy for this thesis, the path management module was placed inside the proxy. However, it does not need to be placed here. The path manager has to have access to information about the queue and the capacity of the different links. This information can be forwarded to the path manager placed elsewhere, and the results from the path manager can be sent back to the proxy. This also makes sense with regard to the time scale the path manager work in compared to the rest of the proxy. Since receiving and sending packets in the proxy are done with microsecond precision, and the path manager only has to do things with millisecond precision, decoupling the two would make sense. Separating a very time-sensitive task and a not-so-time-sensitive task would mean that the hardware each run on can be more optimized for a particular use case. This can give more accurate results and can be something to investigate in the future.

8.1.6 Security

Security is one thing that has not been considered while developing the proxy. Nonetheless, it is a critical aspect of anything that is to be deployed in any network today. Ideally, the proxy should be able to support multiple users and separate the different flows entirely. Encryption should also be supported, though not at the expense where the proxy can not do its work.

Bibliography

[1] Yongmao Ren, Wanghong Yang, Xu Zhou, Huan Chen and Bing Liu. 'A survey on TCP over mmWave'. eng. In: *Computer communications* (2021). ISSN: 0140-3664.

[2] Hongjia Wu, Simone Ferlin, Giuseppe Caso, Özgü Alay and Anna Brunstrom. 'A Survey on Multipath Transport Protocols Towards 5G Access Traffic Steering, Switching and Splitting'. In: *IEEE Access* 9 (2021), pp. 164417–164439. DOI: 10.1109/ACCESS.2021.3134261.

[3] David A. Hayes, David Ros, Özgü Alay and Peyman Teymoori. 'Reliable Consistent Multipath MmWave Communication'. In: *Proceedings of the 24th International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*. MSWiM '21. Alicante, Spain: Association for Computing Machinery, 2021, pp. 149–158. ISBN: 9781450390774. DOI: 10.1145/3479239.3485684. URL: https://doi.org/10.1145/3479239.3485684.

[4] David A. Hayes, David Ros, Özgü Alay, Peyman Teymoori and Tine Margretha Vister. 'Investigating Predictive Model-Based Control to Achieve Reliable Consistent Multipath mmWave Communication'. Elsevier Computer Communications, ACM MSWiM 2021 Special Issue, Submitted, under review. Mar. 2022.

[5] D. A. Hayes, D. Ros and Ö. Alay. 'On the importance of TCP splitting proxies for future 5G mmWave communications'. In: *2019 IEEE 44th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*. Oct. 2019, pp. 108–116. DOI: 10.1109/LCNSymposium47956.2019.9000661.

[6] George R MacCartney, Theodore S Rappaport and Sundeep Rangan. 'Rapid Fading Due to Human Blockage in Pedestrian Crowds at 5G Millimeter-Wave Frequencies'. eng. In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–7. ISBN: 1509050191.

[7] Reza Poorzare and Anna Calveras Augé. 'How Sufficient is TCP When Deployed in 5G mmWave Networks Over the Urban Deployment?' In: *IEEE Access* 9 (2021), pp. 36342–36355. DOI: 10.1109/ACCESS.2021.3063623.

[8] Michele Polese. 'End-to-End Design and Evaluation of mmWave Cellular Networks'. PhD thesis. University of Padua, Nov. 2019. URL: http://paduaresearch.cab.unipd.it/12134/.

[9] Ehab Ali, Mahamod Ismail, Rosdiadee Nordin and Nor Fadzilah Abdulah. 'Beamforming techniques for massive MIMO systems in 5G: overview, classification, and trends for future research'. In: *Frontiers of Information Technology & Electronic Engineering* 18.6 (2017), pp. 753–772. ISSN: 2095-9230. DOI: 10.1631/FITEE.1601817. URL: https://doi.org/10.1631/FITEE.1601817.

[10] A Botta and A Pescapé. 'Monitoring and measuring wireless network performance in the presence of middleboxes'. eng. In: *2011 Eighth International Conference on Wireless On-Demand Network Systems and Services*. IEEE, 2011, pp. 146–149. ISBN: 9781612841892.

[11] Jim Griner, John Border, Markku Kojo, Zach D. Shelby and Gabriel Montenegro. *Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations*. RFC 3135. June 2001. DOI: 10.17487/RFC3135. URL: https://rfc-editor.org/rfc/rfc3135.txt.

[12] M. Kim, S. Ko and S. Kim. 'Enhancing TCP end-to-end performance in millimeter-wave communications'. In: *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. Oct. 2017, pp. 1–5. DOI: 10.1109/PIMRC.2017.8292745.

[13] Michele Polese et al. 'milliProxy: A TCP proxy architecture for 5G mmWave cellular systems'. eng. In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2017, pp. 951–957. ISBN: 9781538606667.

[14] M. Kim, S. Ko, H. Kim, S. Kim and S. Kim. 'Exploiting Caching for Millimeter-Wave TCP Networks: Gain Analysis and Practical Design'. In: *IEEE Access* 6 (2018), pp. 69769–69781. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2880774.

[15] S. Song et al. 'Multipath Based Adaptive Concurrent Transfer for Real-Time Video Streaming Over 5G Multi-RAT Systems'. In: *IEEE Access* 7 (2019), pp. 146470–146479. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2945357.

[16] Alan Ford, Costin Raiciu, Mark J. Handley, Olivier Bonaventure and Christoph Paasch. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 8684. Mar. 2020. DOI: 10.17487/RFC8684. URL: https://rfc-editor.org/rfc/rfc8684.txt.

[17] Michele Polese, Rittwik Jana and Michele Zorzi. 'TCP and MP-TCP in 5G mmWave Networks'. eng. In: *IEEE internet computing* 21.5 (2017), pp. 12–19. ISSN: 1089-7801.

[18] Delia Rico and Pedro Merino. 'A Survey of End-to-End Solutions for Reliable Low-Latency Communications in 5G Networks'. eng. In: *IEEE access* 8 (2020), pp. 192808–192834. ISSN: 2169-3536.

[19] Tobias Viernickel, Alexander Froemmgen, Amr Rizk, Boris Koldehofe and Ralf Steinmetz. 'Multipath QUIC: A Deployable Multipath Transport Protocol'. In: *2018 IEEE International Conference on Communications (ICC)*. 2018, pp. 1–7. DOI: 10.1109/ICC.2018.8422951.

[20] Adam Langley et al. 'The QUIC Transport Protocol: Design and Internet-Scale Deployment'. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 183–196. ISBN: 9781450346535. DOI: 10.1145/3098822.3098842. URL: https://doi.org/10.1145/3098822.3098842.

[21] Z. Krämer, S. Molnár, M. Pieskä and A. Mihály. 'A Lightweight Performance Enhancing Proxy for Evolved Protocols and Networks'. In: *2020 IEEE 25th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. Oct. 2020, pp. 1–6. DOI: 10.1109/CAMAD50429.2020.9209304.

[22] Changsung Lee, Sooeun Song, Hyoungjun Cho, Goeun Lim and Jong-Moon Chung. 'Optimal Multipath TCP Offloading Over 5G NR and LTE Networks'. eng. In: *IEEE wireless communications letters* 8.1 (2019), pp. 293–296. ISSN: 2162-2337.

[23] Salman Saadat, Da Chen and Tao Jiang. 'Multipath multihop mmWave backhaul in ultra-dense small-cell network'. eng. In: *Digital communications and networks* 4.2 (2018), pp. 111–117. ISSN: 2352-8648.

[24] Kun Wang et al. 'On the Path Management of Multi-path TCP in Internet Scenarios Based on the NorNet Testbed'. In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. 2017, pp. 1–8. DOI: 10.1109/AINA.2017.29.

[25] Yeon-sup Lim, Yung-Chih Chen, Erich M. Nahum, Don Towsley and Kang-Won Lee. 'Cross-layer path management in multi-path transport protocol for mobile devices'. In: *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 2014, pp. 1815–1823. DOI: 10.1109/INFOCOM.2014.6848120.

[26] Simone Ferlin, Özgü Alay, Olivier Mehani and Roksana Boreli. 'BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks'. In: *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. 2016, pp. 431–439. DOI: 10.1109/IFIPNetworking.2016.7497206.

[27] Hongjia Wu, Özgü Alay, Anna Brunstrom, Simone Ferlin and Giuseppe Caso. 'Peekaboo: Learning-Based Multipath Scheduling for Dynamic Heterogeneous Environments'. In: *IEEE Journal on*

Selected Areas in Communications 38.10 (2020), pp. 2295–2310. DOI: 10.1109/JSAC.2020.3000365.

[28] Hongjia Wu, Giuseppe Caso, Simone Ferlin-Reiter, Ozgu Alay and Anna Brunstrom. 'Multipath Scheduling for 5G Networks: Evaluation and Outlook'. In: *IEEE Communications Magazine* (Mar. 2021).

[29] *Recommendation ITU-T Z.100 (2021), Specification and Description Language – Overview of SDL-2010*. International Telecommunications Union. URL: https://www.itu.int/rec/T-REC-Z.100-202106-I/en.

[30] Henry Zárate Ceballos et al. *Wireless Network Simulation: A Guide using Ad Hoc Networks and the ns-3 Simulator*. Apress.

[31] *Julia 1.7 Highlights*. URL: https://julialang.org/blog/2021/11/julia-1.7-highlights/.

[32] *Parallel Computing*. Mar. 2022. URL: https://docs.julialang.org/en/v1/manual/parallel-computing/.

[33] *Multi-processing and Distributed Computing*. Apr. 2022. URL: https://docs.julialang.org/en/v1/manual/distributed-computing/.

[34] *Base.Threads.@spawn*. Apr. 2022. URL: https://docs.julialang.org/en/v1/base/multi-threading/#Base.Threads.@spawn.

[35] *ThreadPools.jl*. Apr. 2022. URL: https://tro3.github.io/ThreadPools.jl/build/.

[36] *ThreadPools.@tspawnat*. Apr. 2022. URL: https://tro3.github.io/ThreadPools.jl/build/#ThreadPools.@tspawnat.

[37] *MKL.jl*. Apr. 2022. URL: https://juliapackages.com/p/mkl.

[38] *Base.Channel*. May 2022. URL: https://docs.julialang.org/en/v1/base/parallel/#Base.Channel.

[39] *ConcurrentCollections.DualLinkedConcurrentRingQueue*. May 2022. URL: https://juliaconcurrent.github.io/ConcurrentCollections.jl/dev/#ConcurrentCollections.DualLinkedConcurrentRingQueue.

[40] *ConcurrentCollections.jl: "lock-free" dictionary, queue, etc. for Julia 1.7*. May 2022. URL: https://discourse.julialang.org/t/concurrentcollections-jl-lock-free-dictionary-queue-etc-for-julia-1-7/72501.

[41] *Base.Threads.Atomic*. May 2022. URL: https://docs.julialang.org/en/v1/base/multi-threading/#Base.Threads.Atomic.

[42] *Base.sleep*. Apr. 2022. URL: https://docs.julialang.org/en/v1/base/parallel/#Base.sleep.

[43] *Base.Libc.systemsleep*. Apr. 2022. URL: https://docs.julialang.org/en/v1/base/libc/#Base.Libc.systemsleep.

[44] *clock_nanosleep(2) - Linux manual page*. Apr. 2022. URL: https://man7.org/linux/man-pages/man2/clock_nanosleep.2.html.

[45] *libc(7) — Linux manual page*. URL: https://man7.org/linux/man-pages/man7/libc.7.html.

[46] *Base.Threads.@threads*. URL: https://docs.julialang.org/en/v1/base/multi-threading/#Base.Threads.@threads.

[47] *ConcurrentCollections.ConcurrentDict*. URL: https://juliaconcurrent.github.io/ConcurrentCollections.jl/dev/#ConcurrentCollections.ConcurrentDict.

[48] *Base.Timer*. Apr. 2022. URL: https://docs.julialang.org/en/v1/base/base/#Base.Timer-Tuple%7BFunction,%20Real%7D.

[49] *Libuv*. URL: https://github.com/libuv/libuv.

[50] *sendto(3p) — Linux manual page*. URL: %5Ccite%7Bman-tc%7D.

[51] Paul R. Wilson. 'Uniprocessor garbage collection techniques'. In: *Memory Management*. Ed. by Yves Bekkers and Jacques Cohen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 1–42. ISBN: 978-3-540-47315-2.

[52] *julia/src/gc.c*. May 2022. URL: https://github.com/JuliaLang/julia/blob/master/src/gc.c.

[53] *VirtualBox*. Apr. 2022. URL: https://www.virtualbox.org.

[54] *Introduction to networking Modes*. Apr. 2022. URL: https://www.virtualbox.org/manual/ch06.html#networkingmodes.

[55] *Intel® Ethernet Network Adapter X710-T4L*. Apr. 2022. URL: https://ark.intel.com/content/www/us/en/ark/products/189464/intel-ethernet-network-adapter-x710t4l.html.

[56] *8-Port Multi-Gigabit Ethernet Smart Switch with 2 Dedicated 10-Gigabit Uplink Ports*. Apr. 2022. URL: https://www.netgear.com/business/wired/switches/smart/ms510tx/.

[57] *tc(8) — Linux manual page*. URL: https://man7.org/linux/man-pages/man8/tc.8.html.

[58] *tc-mqprio(8) - Linux manual page*. Apr. 2022. URL: https://man7.org/linux/man-pages/man8/tc-mqprio.8.html.

[59] *Multi-threading changes*. URL: https://github.com/JuliaLang/julia/blob/v1.8.0-beta1/NEWS.md#multi-threading-changes.