Exploring Performance Balancing in a Meshed Satellite MultiWAN Using MPTCP

A Performance Comparative Study of Congestion Control- and Scheduling Algorithms for MPTCP

Master's thesis in Computer Science and Engineering

Katri Lantto
Vera Svensson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Exploring Performance Balancing in a Meshed Satellite MultiWAN Using MPTCP

A Performance Comparative Study of Congestion Control- and Scheduling Algorithms for MPTCP

Katri Lantto

Vera Svensson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Exploring Performance Balancing in a Meshed Satellite MultiWAN Using MPTCP
A Performance Comparative Study of Congestion Control- and Scheduling Algorithms for MPTCP
Katri Lanto
Vera Svensson

© Katri Lantto, 2025.
© Vera Svensson, 2025.

Supervisor: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering
Advisor: Martin Löfgren, Satcube
Examiner: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Exploring Performance Balancing in a Meshed Satellite MultiWAN Using MPTCP
A Performance Comparative Study of Congestion Control- and Scheduling Algorithms for MPTCP
Katri Lantto
Vera Svensson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Improvements to communication protocols has been an active research area with the recent introduction of MultiPath TCP (MPTCP) in the in-tree Linux kernel. This may benefit the area of Satellite Communication as it can be a more reliable and performant alternative in the lossy, high-latency links characterising Satellite Communication. Just like TCP, MPTCP requires a congestion control algorithm to manage its congestion window, which is one of the factors deciding how much data that can be sent on a link at a time. Additionally, MPTCP extends TCPs path manager with a scheduling algorithm which decides on which subflow to send each packet on. This thesis is a performance comparative study on different congestion control- and scheduling algorithms. The research is conducted on a semi-simulated testbed and with portable satellite terminals connecting to GEO satellites providing real-world results, in addition to more easily reproducible tests using an emulated satellite network.

The study revolves around the congestion control algorithms LIA, OLIA, BALIA, and wVegas due to them being readily available with well tested MPTCP implementations. As for packet schedulers, MinRTT, ECF, BLEST, Round Robin, and the currently default scheduling algorithm are tested. The results indicate that some improvements in performance can be made through the choice of congestion control- and scheduling algorithm, especially if tailored to specific network use cases. At the end multiple points for future work and improvements are presented.

Keywords: MPTCP, MultiWAN, GEO, packet scheduling, congestion control, comparative study.

Acknowledgements

We want to thank our supervisor Ahmed Ali-Eldin Hassan for his continuous guidance and encouragement throughout this process. We would also like to express our appreciation to our advisor at Satcube, Martin Löfgren, for providing us with this opportunity. We are incredibly grateful for the time and support he and the entire Satcube team dedicated to answering our questions and assisting us in our problem solving. Finally, we would like to thank everyone who contributed their time by reviewing this thesis.

Katri Lantto, Vera Svensson, Gothenburg, 2025-03-06

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1
1.1 Goals & Challenges . . . 2
1.2 Scope of thesis . . . 3
1.3 Contributions . . . 3
1.4 Outline . . . 3

2 Background 5
2.1 Satellite Communication . . . 5
2.2 Multipath TCP (MPTCP) . . . 6
2.2.1 Scheduling Algorithm (SA) . . . 9
2.2.2 Congestion Control Algorithm (CCA) . . . 13
2.2.3 MPTCP in the Linux kernel . . . 16
2.2.4 Benefits and Use-cases in Satellite Communication . . . 17
2.3 Related work . . . 17

3 Experiment Design 19
3.1 Selection Criteria . . . 20
3.2 Testcases . . . 21
3.3 Data Collection and Result Analysis . . . 22

4 Implementation 25
4.1 Testbeds . . . 25
4.1.1 Parental Board (PB) . . . 25
4.1.2 Network Switch . . . 27
4.1.3 Network Emulation (Jalapeño) . . . 28
4.2 MPTCP Setup . . . 29
4.2.1 Subflows and Routing . . . 29
4.2.2 Congestion and Scheduler Configuration . . . 29
4.2.3 Encapsulating packets on Live Testbed . . . 30
4.3 Performance Evaluation Tools . . . 31
4.3.1 iPerf3 . . . 31
4.3.2 Sockperf . . . 32

4.3.3 Sar 33
4.3.4 Wireshark 33
4.4 Algorithm Details 34
4.4.1 Congestion Control Algorithm Implementation 34
4.4.2 Scheduling Algorithm Implementation 34

5 Results **39**
5.1 Reference performance of testbeds 39
5.2 Performance Comparison of Congestion Control Algorithms 41
5.3 Small Differences in Performance of Scheduling Algorithms 48
5.4 Varying Results Between Model and Live Testbeds 48
5.5 Validity of Latency Measurements 55
5.6 Validity of Throughput and Goodput measurements 57

6 Conclusion **59**
6.1 Notable Results 59
6.1.1 Concerns Regarding SATCOM Network Emulation 60
6.1.2 The Limitations of Live Tests 60
6.2 Possible Improvements 61
6.2.1 Additional testcases: 62
6.2.2 Future Work 63
6.3 Final Words 63

Bibliography **65**

List of Figures

2.1 MPTCP handshakes for establishing connection, and adding subflow [4]. . . . 8

4.1 Overview of the Model testbed. . . . 26
4.2 Overview of the Live testbed. . . . 26
4.3 Visual representation of the implemented subnet separation on the testbeds. . . . 27

5.1 Mean throughput per CCA (left) and per SA (right), with standard deviations, from aggregation of iPerf3 results on the Live Testbed (top) and Model Testbed (bottom). The baseline marks the mean throughput of Cubic and MinRatio respectively. . . . 42
5.2 Mean throughput, with standard deviation, from aggregation of iPerf3 results on the Live Testbed. The baseline marks the mean throughput of Cubic_MinRatio 43
5.3 Mean latency, with standard deviations, from aggregation of Sockperf under-load results on the Live Testbed. The baseline marks the mean latency of Cubic_MinRatio 44
5.4 Average latency per CCA (left) and per SA (right), with standard deviations, from aggregation of Sockperf under-load results on the Live Testbed (top) and Model Testbed (bottom). The baseline marks the average latency of Cubic and MinRTT respectively. . . . 45
5.5 Mean percentage of detected Retransmissions (lighter, skinnier bars) and Out-of-Order packets (darker, thicker bars) by Wireshark for L.Def and L.Het with the dotted line showing the mean percentage of Out-of-Order packets for Cubic_MinRatio 46
5.6 Average goodput per test scenarios L.Def and L.Het shows tests, mostly combinations utilising BALIA reaching the dotted baseline which marks the mean performance of the two subflows when running combination Cubic_MinRatio 47
5.7 Aggregation of average goodput [Byte/s] divide per subflow for all test types per algorithm combination (large is better) when running iPerf3 on model testbeds. . . . 49
5.8 Mean latencies from aggregation of Sockperf under-load results on the Model vs Live testbed. Sorted by the Live testbed latencies in descending order. . . . 51

5.9 Plot of Send Window size per time unit for 5 test runs utilising the baseline scheduler MinRatio shows how well the CCA maintains a high send window (large is better). The results shown are as sent from iPerf3. Different colours are representative of different IP sources, with Live testbed having three due to the WireGuard configuration. . 52
5.10 Plot of average transmitted Bytes/s for L.Def and M.Def shows that Live testbed (top) has a more even distribution of throughput throughout the test, while Model (bottom) testbed shows a sharp decline after the initial Slow Start. Data measured through SAR data on the network interface. Dotted line shows the theoretical bandwidth capacity. 53
5.11 Mean percentage of detected Retransmissions and Out-of-Order packets by Wireshark for L.Def and M.Def shows that Live test bed has a higher level of packets marked as Out-of-Order (darker, thicker bars) than Model testbed while retransmissions (lighter, skinnier bars) remain comparable. 54
5.12 Mean latencies, with standard deviations, from aggregation of 20 Sockperf under-load tests per testcase on the Model Testbed. The baseline marks the mean latency of Cubic_MinRatio 55
5.13 Mean throughput, with standard deviation, from aggregation of iPerf3 results on the Model Testbed. The baseline marks the mean throughput of Cubic_MinRTT. 56

List of Tables

2.1 Height of GEO/LEO satellites and commonly used frequency bands [11], [13]. 6
2.2 Relevant information used for MPTCP connections as they are stored within the confines of IP/TCP stack. 7
2.3 Relevant flags for MPTCP set in packet header [15], [16]. 8
2.4 Relevant options for MPTCP set in packet header [4]. 9
2.5 Overview of Scheduling Algorithms for MPTCP. 10
2.6 Properties checked by active vs. available check. 13
2.7 Comparison of included features in the MPTCP versions 0.96, 5.15.60 and latest. 16

3.1 Default network conditions in controlled tests. 21

4.1 Technical details of Parental Board with its mounted System-on-Module. 26
4.2 Configuration of network switch. 28
4.3 Technical details of network emulator board (Jalapeño). 29
4.4 Overview of Key Performance Indicators and tools used. 31

5.1 Mean results for TCP using Cubic from Sockperf under-load results. Latency is computed from Sockperf's logs while throughput the transmitted kB/s measured by SAR. 40
5.2 Mean throughput for MPTCP using Cubic from SAR measurements during Sockperf under-load results. 41

1
Introduction

In remote or disaster-stricken regions, a fast and reliable internet connection is crucial for applications such as humanitarian aid, military operations, and media communications. Traditional cellular networks are often inaccessible in these areas due to the lack or destruction of essential infrastructure like base stations (BS). In these cases portable satellite terminals present a viable solution by enabling connectivity through either geostationary (GEO) or low-earth (LEO) orbit satellites. These terminals establish connections via satellite beams, unidirectional streams of radio waves, that carry data between the terminal and satellite [1].

However, while satellite communication (SATCOM) can substitute the more common cellular or wired networks, it also brings unique challenges. This is due to the physical properties of satellite links, being sent over radio-waves over massive distances. Thus, common issues with networks such as packet jitter and drops, lack of memory, and poor scheduling are heightened as SATCOM connections are often characterised by limited bandwidth, high latency, and being error-prone [2]. Through the common Transmission Control Protocol (TCP) only one beam can be used per connection, leaving them extra vulnerable to obstruction which are often the cause of SATCOM errors, leading to packet loss.

Such errors can be mitigated through utilising multiple connections simultaneously. Additional network links could include diverse link types through different beams and satellite types or even through a combined network also utilising cellular or wired connections [3]. This can both increase bandwidth, reduce network downtime, and introduce the possibility of redundant scheduling, i.e. scheduling packets multiple times from the start.

In order to support the control of multiple links for the same session, traditional TCP is insufficient as it is designed for single-path communication. Instead, a MultiWAN solution must be used, such as the MultiPath TCP (MPTCP) protocol designed by the Internet Engineering Task Force (IETF) to be an extension of TCP. MPTCP allows for simultaneous use of multiple network paths [4]. The MPTCP protocol modifies the TCP handshake through added header options to communicate the end-points communication capabilities. Further it builds out the path manager to enable connecting multiple sub-connections, or subflows, through the header options. New packet scheduling (SA) and congestion control (CCA) algorithms have been introduced to handle the specific issues that come with MPTCP networks to be used in conjunction with the protocol [5].

To explore the effectiveness of MultiWAN using MPTCP in SATCOM, this thesis investigates how the latter components, the SA and CCA, affect performance under the constraints of typical satellite communications. For example, it is interesting to explore how bandwidth aggregation across subflows could enhance throughput for large data transfers, while redundant scheduling might help prevent retransmission on the high-latency SATCOM links [6]. One of the main considerations when touching upon SATCOM is the handling of the heterogenous links and higher level of packet loss common in satellite network, due to the increased time which recouping takes [3], [7].

Further as MPTCP is of a more complex nature than TCP, extra considerations must generally be made within the algorithms. A large source of extra complexity comes from handling the congestion window, where extra leniency could be implemented in order to not lower the congestion window for every loss, since satellite networks suffer from high packet loss [3]. Similarly, in-order packet scheduling over heterogenous links requires one to account for the differences in latency when choosing subflow [7], [8]. Such scheduling could reduce jitter, or differences in delay, which is not only based on links characteristics but is also affected by lost packets and re-ordering of packets [6].

Although multiple studies of MPTCP exists in different network environments, such as wired, cellular, and LEO satellite networks, the research on its application within GEO networks remains limited. However, a 2023 Chalmers thesis explored the performance of the standard implementation of MPTCP with mixed-link environments containing GEO links. They found that GEO links could gain the benefits of aggregated bandwidth. Further they noted, as other research papers, that homogeneous links gained more benefits from MPTCP overall [9]. Building on that, this study will implement and test known CCAs and SAs within the Linux kernel and compare performance metrics in a GEO-only SATCOM environment. The results aim to identify configurations which improve SATCOM network performance.

1.1 Goals & Challenges

The primary objective of this thesis was to conduct a comparative study of Congestion Control and Scheduling Algorithms in a meshed, SATCOM MultiWAN. The study explores how various CCAs and SAs can improve MPTCP performance over GEO satellite links and outlines key performance trade-offs.

To evaluate the algorithms, the following metrics were focused on:

- **Throughput**: A fundamental metric for efficient and responsive network communication most improved by effective congestion handling.
- **Latency**: A metric reflected in the average response time of the network, improved by effective scheduling over faster subflows, with reduction of re-ordering and packet losses.
- **Out-of-order (OFO) packets**: OFO packets could lead to head-of-line blocking, leading to lost packets due to buffer bloat effectively, reducing useful data

transmissions and increasing response time as packets must be re-ordered or retransmitted.

This thesis thus aims to determine if the chosen SAs and CCAs can achieve improved MPTCP performance in GEO SATCOM networks, specifically focusing on increased throughput, effective path utilization, and improved congestion management.

1.2 Scope of thesis

The main limiting of the scope of this thesis was the exclusive focus on geostationary (GEO) satellite environment which was chosen due to its unique latency and bandwidth constraints, as well as the lack of comparable studies in the area. Although low-earth orbit (LEO) satellites provide an interesting addition to SATCOM networks, they involve different terminal requirements which would increase the scope of the thesis by a large margin.

Further, the study was limited to the implementation and testing of algorithms within a specific version of the Linux kernel, due to the chipsets used in testing. This contributes to the choice of exclusively focusing on the Multipath TCP implementation as it is available in the Linux kernel and is open source. This focus on real-world SATCOM testing rather than simulation aims to produce practical results applicable to multipath SATCOM systems. To simplify the focus, only a subset of CCAs and SAs are tested, as is discussed in Chapter 3.

1.3 Contributions

The research for this thesis was conducted both in semi-simulated testbeds and with physical satellite terminals connecting to GEO satellites. These terminals, provided by Satcube were set to communicate with a server by transmitting data to GEO satellites which relayed the information back to the satellite terminal passing information back, finally reaching the destination server. Tests performed on a testbed, simulating network delays and environment such as disturbances, provide easily reproducible results; meanwhile, live tests allow for evaluation of performance in a realistic environment which better represents variabilities such as link speed and environmental conditions.

As such we conducted a performance comparative study of Scheduling and Congestion Control Algorithms for MPTCP performed on satellite terminals connected through GEO satellites to a server on the ground. Through the results presented, we hope to benefit the field of satellite communication as well as MultiWAN networking.

1.4 Outline

This report provides a brief introduction to MPTCP and SATCOM fundamentals along with related studies of scheduling and congestion control algorithms, all of which is presented in Chapter 2. Chapter 3 describes our experiment design and

analysis methods while 4 details the tools and testbed alongside further information on the implementation of the chosen algorithms. The results are presented in Chapter 5, followed by a discussion and conclusions in Chapter 6.

2

Background

This chapter will present theory on relevant areas to the thesis. Firstly details of satellite communication are explained, including the relevant bands and frequencies in which they operate as well as their height - both of which influence the propagation delay of links. Next an overview of Multipath TCP is given explaining details which are important to understand or know of in order to understand decisions and implementations of this thesis. This also includes a presentation of packet Scheduling Algorithms (SA) and Congestion Control Algorithms (CCA) work as well as details to some algorithms which we have looked at. Lastly a brief introduction to relevant parts of the implementation of MPTCP in the Linux kernel is given. The chapter concludes with some notes on related work.

2.1 Satellite Communication

Satellite communication is done through ground stations which can transmit and receive data wirelessly through relaying the data to satellites in earth orbit [2]. The communication is done through satellite beams which are uni-directional flows of radio-waves. A satellite has multiple frequency bands which earth stations can connect to simultaneously. In order for uplink and downlink to and from the satellite to not interfere with each other they operate in different frequency bands. An uplink/downlink pair to and from a satellite is called a channel. Duplex links can be formed through utilising two channels to establish a circuit between two earth stations.

Satellite terminals, such as the Satcube Ku [10], are portable versions of ground stations used for telecommunication. Their main component is their antenna which transmits the data through the same beams which ground stations can connect to. Their use enables communication in areas where cellular networks are unavailable, such as remote locations or catastrophe stricken areas, even if they lack a permanent satellite dish. As the name implies the Satcube Ku functions in the Ku frequency band at frequencies between $10.7 - 18$ GHz, although it is possible to connect to different bands [11]. Some common bands can be seen in Table 2.1 where communication within lower frequencies such as the C band are generally more robust to weather interference while being slower and requiring a larger disk, while higher frequency bands such as the Ka band is faster but susceptible to interference. The Ku band operates between these bands and will generally have downlink set in a

higher frequency than uplink, allowing for greater download speeds.

There are several types of satellites that can be used for communication. They are categorised by orbit, with most common satellites for communication being those in geostationary orbit (GEO) and low-earth orbit (LEO) [12], [13]. The height of orbit alongside the frequency with which radio-waves are sent defines the propagation delay of the link [2]. Using the link characteristics noted in Table 2.1 alongside the speed of light, one can derive the delay of GEO channel to be ~ 260 ms while LEO boasts of a delay of just ~ 10 ms.

GEO height	**LEO height**	**C band**	**Ku band**	**Ka band**
~ 36000 km	$160 - 2000$ km	$4 - 8$ GHz	$12 - 18$ GHz	$26 - 40$ GHz

Table 2.1: Height of GEO/LEO satellites and commonly used frequency bands [11], [13].

Satellites in geostationary orbit have some leverage in coverage over LEO satellites due to their orbit, however, it comes at the cost of higher delays. As the name suggests GEO satellites are stationary in the sky as their orbital speed is matched to earths rotational speed. Due to the height required for this, they also have a higher footprint i.e. coverage. This means that antennas do not have to be re-pointed or follow the satellites in the sky. Further selection of a GEO satellite can be done automatically when the earth stations latitude and longitude is known. LEO satellites on the other hand orbit earth many times per day, hence going in and out of reach from the ground frequently. However, it is important to note that the stationary property of GEO satellites means that they can only orbit around the earth's equator, leading to longer propagation delays the closer one gets to the earths poles [2].

For most uses, connection to cellular networks such as 4G and 5G will be most useful. However, in remote areas with no or poor infrastructure due to distance from society, or destruction such as war stricken areas, that is often not possible. This leads to some use cases of SATCOM which are out of the norm of network usage, perhaps not favouring high throughput and low latency as much as everyday use of the internet. Examples of such work could be critical communication for humanitarian aid, military operations, and media communications. Especially live video transmissions for media communications, such as news outlets, favour reliably high latency while not requiring the highest throughputs.

2.2 Multipath TCP (MPTCP)

Multipath TCP (MPTCP) is an extension of normal TCP, or Single Path TCP, which makes it possible to utilise several TCP connections concurrently. The implementation details are defined in the standard `RFC 8684` proposed by the Internet Engineering Task Force (IETF) [4]. The protocol is designed to be used by legacy applications with no changes. This is possible through converting sockets from which applications communicate from TCP sockets to MPTCP sockets, without needing to

give the application itself any additional context. However, applications can request MPTCP operation. It is only the hosts of the path which handle MPTCP headers, so intermittent nodes along the path function exactly the same as for TCP. As such, MPTCP mainly operates in the transport layer of the IP/TCP networking stack as shown in Table 2.2, aiming to be transparent to both higher and lower levels.

IP/TCP Stack	**Operation**	
Application	Data & Session info	
Transport	MPTCP (End-to-End connection)	
	TCP (Subflow)	TCP (Subflow)
Internet	IP	IP
Link	-	-

Table 2.2: Relevant information used for MPTCP connections as they are stored within the confines of IP/TCP stack.

In order to start MPTCP connections a modified TCP handshake is used as seen in Figure 2.1 [14]. The handshake is functionally the same in order to keep the transparency to other levels. However, a flag `MP_CAPABLE` is added to the `SYN` and `SYN-ACK`. Additionally, the `initial ACK` and `data` packets also carry the `MP_CAPABLE` option. This option carries flags which allow the hosts to exchange information necessary to authenticate new subflows [4]. If the option is not set from the receiver then the connection is downgraded to a TCP connection.

Multiple subflows are added through use of the authentication keys set with the `MP_CAPABLE` options. Additional subflows begin in the same way as a regular TCP connection, however, carrying the `MP_JOIN` option. Details to both of these can be found in Table 2.4. The sender can also choose to instead simply advertise new addresses through `ADD_ADDR`, thus allowing the receiver the option of establishing the connection.

In addition to the establishment of connection and new subflows the sender is also responsible for path management, packet scheduling, and congestion control [4]. However, in order to not flood the receiver with too much data, or unnecessarily starving it from data, they communicate using flags set through the header according to the TCP protocol defined by `RFC 9294` [15]. The congestion control responsible for amount of data sent mainly makes use of `CWR, ECE, PSH` flags. These flags, and other, can be seen in Table 2.3. The `ECE` flag introduced in `RFC 3168` can account for congestion before it has lead to dropped packets [16]. The `CWR` flag supports this implementation through notifying connections of the prematurely lowered congestion window. Although this is not used in all implementations of MPTCP and requires support from the IP layer of the networking stack. The `PSH` flag, however, is more commonly used [15]. It indicates that data packets should be directly transmitted instead of queued in order to better fill out control windows. It can also be used in situations which require sending of data to avoid deadlocks.

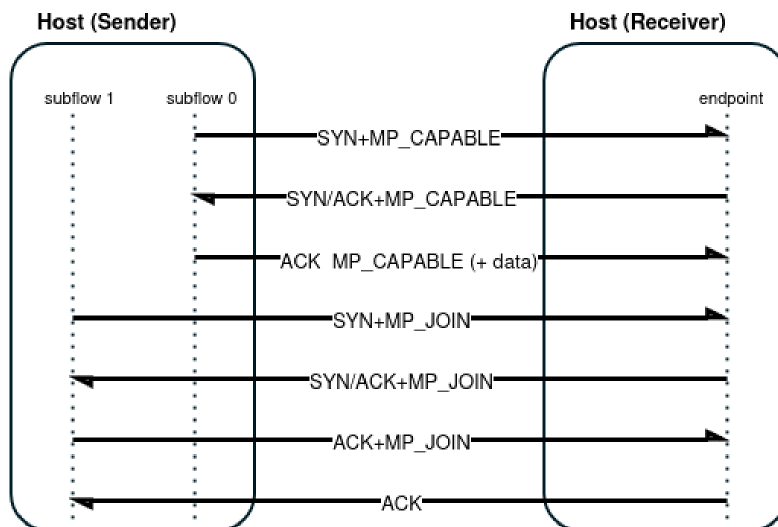

Figure 2.1: MPTCP handshakes for establishing connection, and adding subflow [4].

The communication through `ACK` and `SYN` flags lie at the core of the TCP protocols. This continues to be true for MPTCP as it expands on the use of Sequence Numbers (SN) into Subflow Sequence Number (SSN) and Data Sequence Number (DSN), the latter of which is equal to the TCP implementation in function in that it creates a sequencing of the original data as it should be reassembled from the subflows [4], [15]. As mentioned it is through these SNs that packets are reordered and also how packet drops are noticed. Further, it is through `ACK/SYN` packets which new subflows are advertised or added by the path manager through the `MP_JOIN` options.

Bit offset (0-16)	Flag	Explanation
8	`CWR`	Congestion Window Reduced
9	`ECE`	Explicit Congestion notification Echo
11	`ACK`	Acknowledgement field is significant
12	`PSH`	Push function
13	`RST`	Reset the connection
14	`SYN`	Synchronise sequence numbers
15	`FIN`	No more data from sender

Table 2.3: Relevant flags for MPTCP set in packet header [15], [16].

Proper closure of communication is important in order to not keep hosts waiting for new packets after a flow of data has been completed. This is communicated through setting the `FIN` flag [15]. However, there are more cases in which a connection should be closed. Abrupt closure of a connection (TCP or MPTCP subflow) is done through the `RST` or `MP_TCPRST` flag [4]. In order to abruptly close all MPTCP subflows of a session the `MP_FASTCLOSE` flag should be used as seen in Table 2.4. This can be done more gracefully by setting the subflow to a backup flow, to start redirecting data, through the `MP_PRIO` flag, and lastly removing the address through `REMOVE_ADDR`.

Bit offset	Flag	Explanation
0	MP_CAPABLE	Advertises multipath capabilities present in SYN/ACK
1	MP_JOIN	Request of connection to join as subflow present in SYN/ACK
2	DSS	Data Sequence Signal (Data ACK and sequence mapping)
3	ADD_ADDR	Announcement of additional addresses or ports to reach host through
4	REMOVE_ADDR	Advertisement that an address has become invalid or shouldn't be used
5	MP_PRIO	Used to change subflow priority
6	MP_FAIL	Used to communicate MPTCP format and subflow checksum errors
7	MP_FASTCLOSE	Extends RST which in MPTCP to close all subflows
8	MP_TCPRST	Used to reject MP_JOIN and duplicate RST behaviour
9	MP_EXPERIMENTAL	

Table 2.4: Relevant options for MPTCP set in packet header [4].

2.2.1 Scheduling Algorithm (SA)

The scheduling algorithms (SAs) task is to decide on what subflow each packet is to be sent. Throughout the evolution of MPTCP the default scheduler has changed. In research papers, the Minimum Round-Trip-Time first (minRTT) SA is often referred to as the default MPTCP scheduler [17], this is due to it being used as the default in the mature pre-release MPTCP implementation (v0.96) [18]. As the name implies, it schedules packets to the subflow with smallest RTT until it is congested. However, the scheduler used as default in v5.15 and latest in-tree MPTCP implementation instead selects subflows with the lowest available send window to total window ratio [19], [20]. It is referred to as the minRatio SA in this thesis. These, and other suggested scheduling algorithms, can be seen in Table 2.5 which contains an overview of their approaches for quick comparison. They are further described in the paragraphs below, which categorise the algorithms based on their scheduling strategy.

Scheduling based on minimum Round-Trip Time (RTT) is in its most basic form the old default algorithm which simply selected the subflow with lowest RTT [17], [18]. This can cause Head-Of-Line (HOL) blocking, as the faster subflows must wait for higher-delay subflows packets to arrive to release packets from the Out-of-Order (OFO) queue. This causes *burstiness* in the data stream as delivery is delayed until the latter packets arrive. This buffering is directly related to the congestion window as packets arriving to a full queue will be dropped, causing a congestion window reduction. As it can not be entirely avoided applications must appoint enough buffer space for subflows to be utilised properly [21].

To avoid this, minor changes have been suggested in minRTT/RP and minRTT/BM with focus on *Retransmission and Penalisation* and *Bufferbloat Mitigation* respectively [29]. The first tries to compensate for delay differences through opportunistic retransmissions which re-inject segments causing HOL blocking on other subflows which have space remaining in their congestion window. Penalising slow subflows through reduction of their congestion window, it attempts to proactively avoid HOL blocking. *Bufferbloat mitigation* on the other hand can utilise the effects of *Retransmission and Penalisation* to effectively utilise available buffers. This is done through adaptively setting the size of buffers, which is proposed to be done through capping

SA	Approach	Use-case	Implementation
minRTT [17]	Selects available subflow with the minimum RTT.	Heterogenous network	out-of-tree
minRTT/RP [21]	minRTT with Retransmission and Penalisation	General	out-of-tree (v.089)
minRTT/BM [21]	minRTT with bufferbloat mitigation	General	out-of-tree (v.089)
minRatio [19]	Selects active subflow with the minimum available window to total window ratio.	Heterogenous network	in-tree (default)
Earliest Completion First (ECF) [22]	Attempts to achieve in-order arrival by using RTT estimation, current CWND and send buffer.	Video streaming	in kernel (v0.96)
Largest Window Space (LWS) [23]	Selects subflow with largest CWND	Bulk transfer	algorithm in paper
Lowest Time/Space First (LTS) [23]	Selects subflow with lowest ratio of latency to window space	Bulk transfer	algorithm in paper
Estimated Subflow Path Capacity (ESPC) [24]	Estimates path capacity as the mid point between latest CWND before a loss and a minimum value of currently outstanding packets, effectively decreasing performance loss due to slow-start	Bulk transfer	algorithm in paper
Blocking Estimation (BLEST) [25]	Estimates if a path would cause head-of-line blocking and dynamically adjusts scheduling	Heterogeneous networks	out-of-tree (v0.95)
Out-of-order Transmission for In-order Arrival (OTIAS) [8]	Estimate packet arrival time on subflows and schedules packets for in-order arrival	Real-time applications	algorithm in paper
Delay Aware Packet Scheduling (DAPS) [26]	Schedules chunks of data according to RTTs and sequence numbers for in-order arrival	General	algorithm in paper
Round-Robin (RR) [17]	Schedules data on available subows in round-robin fashion	Academic/ testing	out-of-tree
Weighted RR (WRR) [27]	RR with weighted subflows	Genral or bulk transfer	algorithm in paper
Redundant [17]	Sends data on all subflows, improving latency by sacrificing bandwidth	Academic/ testing or industrial systems	out-of-tree
Highest Send Rate First (HSR) [23]	Selects subflow with highest send rate	Bulk transfer	algorithm in paper
Shortest Transfer Time First (STTF) [28]	Selects subflow with shortest estimated transfer time	Web and interactive apps	algorithm in paper

Table 2.5: Overview of Scheduling Algorithms for MPTCP.

the congestion window to the value at the point when smoothed RTT is twice as large as base RTT [29]. This is helpful as maintaining small buffers is less costly than maintaining large buffers, however, the *Retransmission and Penalisation* keeps the CWND of subflows artificially low [25].

Earliest Completion First (ECF) is a scheduling algorithm which combines the use of RTT and congestion window (CWND) which bases the space available in CWND in multiples of RTT [22]. The algorithm checks the fastest subflow, and if it does not have sufficient space in the CWND, moves onto the second fastest subflows and calculates whether it is worthwhile to wait for an opening in the CWND for the fastest subflow. If not, it will schedule it on the second fastest subflow, or any good subflow based on minRTT. This algorithm is good at handling asymmetric i.e. heterogenous paths and aims to avoid having fast subflows waiting for packet delivery from slower subflows due to Out-of-Order buffering [28].

Scheduling based on congestion window (CWND) lays more focus on the CWND than RTT, although both are often utilised as seen above. One algorithm which plainly focuses on the CWND is the new default algorithm which we denote as minRatio [19], [20]. It selects the active subflow based on minimum occupied windows space in relation to total window space. Other algorithms such as ESPC, LWS, LTS, and BLEST have found different ways to connect CWND to RTT in order to make the scheduling implementation more robust. The main point of the algorithms is that the disconnect between CWND and RTT quickly leads to HOL blocking which has long-term impacts leading to poor bandwidth aggregation [25]. BLEST or Blocking Estimation-based MPTCP scheduler does so by assuming how long a segment will occupy space in the main MPTCPs send window and estimates the amount of data in flight. The remaining space is allocated to the fastest subflow and is updated for each RTT, as it generally grows in congestion avoidance.

LTS (Lowest Time per Space) and LWS (Largest Window Space) algorithms calculate the available space on a subflows CWND as the difference between the size of outstanding packets in the subflow and its current CWND [23]. LWS simply selects the subflow with the largest available space while LTS uses it in conjunction with the paths RTT and selects the path with the lowest ratio of RTT over available space. However, it is possible for the available space to be negative at which point both of the algorithms, as many function as the minRTT scheduler. ESPC or Estimating available Path Capacity scheduler instead estimates the available capacity similarly to how the TCP congestion control sets CWND in the congestion avoidance phase as can be read about under Section 2.2.2 [24]. It utilises the *Additive Increase and Multiplicative Decrease* in a binary search tree to select subflows with can schedule packets without causing congestion. Presented with multiple subflows with available space it uses the minRTT algorithm to pick one. This is a more precise, although more computationally heavy, way to achieve the effect of LWS and keeps the paths near their saturation point longer before causing decrease in the CWND size.

Scheduling for in-order arrival DAPS or Delay-Aware Packet Scheduling creates a schedule based on the RTT of multiple paths, splitting the data to send over different subflows according to the ratio of their RTT [26]. This sends a suitable amount of data on faster subflows if their CWND allows but schedules remaining packets on slower subflows. It instantly schedules the subsequent data fit into the next CWND on the slower path, hopefully making them arrive in-order. The main goal not being reduced RTT but efficient use the Out-of-Order buffer. OTIAS or Out-of-Order Transmission for In-Order Arrival scheduling calculates the available space just like LWS as the difference between CWND and outstanding packets and uses it with the size of waiting packets to estimate the one-way transmission delay or wait time [8]. It selects the fastest estimated subflow and schedules as many packets as possible on it. It permits queueing data on subflows without available CWND [28].

Scheduling based on calculated transmission time is done on the ECF scheduler, however, there are multiple other implementations of such an algorithm. STTF or Shortest Transfer Time First is similar to the ECF and OTIAS implementations in that it estimates transfer time of segments for each subflow, although it will send all unsent data on the fastest subflow [28], [30]. The distinction being that it traverses all unsent segments and compares the transfer times per subflow, selecting the subflow with the shortest transfer time for the segment. It does not as many other algorithms assume that the connection is in the slower increasing *Congestion Avoidance* phase of congestion control (Section 2.2.2) but also considers *Slow Start* stage which doubles the CWND per transmission until failure. As such it achieves much greater throughput for connections with shorter flows or bursts of data.

Round Robin (RR) scheduling is in its own class as it is known to be a bad algorithm to use in networks but is known to be a bad algorithm as stated on MPTCP website [17]. RR scheduling does not attempt to improve latency or throughput and simply transmits packets through subflows in-order, skipping subflows only if they are unavailable. An extension to the RR algorithm is the Weighted Round Robin (WRR) which add priority to some subflows [27]. This does not improve upon performance any significant amount.

Redundant scheduling is also in its own class since it does not attempt any improvements based on mentioned metrics. It simply schedules all packets on all available subflows redundantly. It is known to be a bad algorithm to use in networks and is only recommended for academic or research purposes.

Active vs. Available subflows: The schedulers differ in what metrics they consider and prioritize when making their scheduling decisions. However, a recurring step is to check if a subflow is active or available [18], [19]. These two checks state requirements that the subflow must meet to allow messages to be scheduled on them. Requirements differ between schedulers, hence some use the active check and some the available check. The active check is less strict and is a subset to the available check, as can be seen in Table 2.6. A subflow is active if it is open, isn't stale, and

is fully established if it is joining a MPTCP connection [19]. Further if the subflow is active, has space in its send window and a receiver which is multipath capable, it is classified as available.

Subflow requirements	**Active**	**Available**
Not stale	✓	✓
Open	✓	✓
Fully established if joining	✓	✓
Receiver is multipath capable		✓
Send window is not full		✓

Table 2.6: Properties checked by active vs. available check.

2.2.2 Congestion Control Algorithm (CCA)

The congestion control algorithm (CCA) aims to enhance throughput and fairness whilst preventing delay and packet loss. Most CCA, including those researched in this thesis, are carried out in the congestion avoidance phase with different strategies for additive-increase/multiplicative-decrease (AIMD) [31]. Meanwhile, other TCP/MPTCP algorithms such as slow start, fast retransmissions, and fast recovery generally follow the standard TCP implementation. These AIMD strategies should follow MPTCP:S goals which is to achieve [5]:

1. **Improve Throughput:** Improve or maintain throughput compared to the best available single path flow.
2. **Do no harm:** Do no harm to shared paths.
3. **Balance congestion:** Balance congestion by moving as much traffic as possible off its most congested path.

There are two types of CCAs: coupled and uncoupled [32]. For coupled CCAs all subflows share the same congestion window (CWND), whilst for uncoupled CCAs each subflow has independent CWNDs. Uncoupled CCAs or used for normal TCP connections, such as Cubic [33]. However, when used with MPTCP connections it may starve regular TCP connections when competing at a shared bottleneck, thus breaking the goal 2 - "do no harm to shared paths". To fulfil the three goals, these four uncoupled CCAs has been suggested:

- Linked Increase Algorithm (LIA) [5]
- Opportunistic LIA (OLIA) [34]
- Balanced Linked Adaptation (BALIA) [35]
- weighted Vegas(wVegas)[36]

Each coupled CCA is described in further detail under its respective paragraph below. For future equations, let the variables below denote the following characteristics of the MPTCP network:

- Set of all subflows: I
- A subflow in the set: i
- Total congestion window: cwnd
- Receiver advertised window: rwnd
- Amount of outstanding data in network: Flight Size
- Aggressiveness (increase rate): α
- Loss rate of flow: p_i
- Round-trip time of flow: rtt_i
- Max segment size of flow: MSS_i
- Send size threshold: ssthresh

LIA, OLIA and BALIA share some of the general functionality which is also used by the standard uncoupled TCP CCAs [5], [34], [35]:

- **Slow start:** Slow start is used when the slow start threshold is greater than the congestion window `ssthresh > CWND`, otherwise congestion avoidance is used [37]. This is primarily at the beginning of a transfer. It probes the network by slowly increasing the CWND to figure out its capacity whilst avoiding congestion. For each ACK the CWND is increased by $\min(N, SMSS)$, where `N` is the number of previously unacknowledged bytes acknowledged in the incoming ACK and `SMSS` is the senders MSS which provides an approximate increase of CWND with a full size segment per RTT.

- **Congestion avoidance:** Congestion avoidance is used while no congestion is detected and `ssthresh < CWND` [37]. It increments the CWND once per RTT by at most SMSS.

- **Fast retransmit/recovery:** The fast retransmit algorithm is used to detect that a segment has been lost and quickly retransmit the lost segment without having to wait for the retransmit timer [37]. The algorithm deems a segment to be lost if it receives 3 duplicate ACKs. After the retransmission, the fast recovery algorithm is used until a non-duplicate ACK is received. The fast recovery algorithm essentially entails that congestion avoidance, and not slow start, is used. Meaning the CWND in increased by `SMSS` for each additional duplicate ACK.

Linked Increase Algorithm (LIA): LIA aims to be fair and move traffic away from congested links by changing the size of CWND when the system experiences packet losses. Decreasing the CWND follows the same process as normal TCP (2.1), while increasing CWND is done according to an increase formula (2.2) when receiving an ACK [5], [32]. Larger total *cwnd* in relation to α leads to smaller increases. The second argument of formula (2.2) $\frac{1}{\text{cwnd}_i}$ represents what a SPTCP link would be granted and creates an upper limit of the allowed increase. This limit is done to adhere to the second goal of MPTCP.

$$\frac{\text{cwnd}_i}{2} \tag{2.1}$$

$$\min(\frac{\alpha}{\text{cwnd}}, \frac{1}{\text{cwnd}_i})$$
$$\text{where} \quad \alpha = \text{cwnd} * \frac{\max_{r \in I}(\text{cwnd}_r \,/\, \text{rtt}_r^2)}{(\sum_{k \in I}(\text{cwnd}_k \,/\, \text{rtt}_k))^2} \tag{2.2}$$

Opportunistic Linked Increase Algorithm (OLIA): OLIA is a modification of LIA that aims to provide both good congestion balancing and responsiveness simultaneously [34]. This is done by altering the increase step of the CWND, while keeping the same decrease strategy, seen in Equation 2.1. For each ACK on subflow i the increase step is performed as per (2.3), with the calculation of α_i dependent on the subflows current CWND and expected performance in comparison to other subflows. α_i controls the aggressiveness of the CWND increase and guarantees the algorithms responsiveness and stability.

$$\left(\frac{\text{cwnd}_i \,/\, \text{rtt}_i^2}{(\sum_{k \in I}(\text{cwnd}_k \,/\, \text{rtt}_k))^2} + \frac{\alpha_i}{\text{cwnd}_i} \right) * \text{MSS}_i * [\text{Acked Bytes}] \tag{2.3}$$

Balanced Linked Adaptation (BALIA): BALIA is a modification of LIA and OLIA that aims to balance the trade-offs between friendliness and responsiveness [35]. It alters both the increase and the decrease CWND steps so that the increase is performed as per (2.4) and decrease as per (2.5) with α_i as the aggressiveness parameter for that subflow. In the case of only one subflow, $\alpha_i = 1$, BALIA is reduced to the uncoupled CCA TCP Reno.

$$\frac{\text{cwnd}_i \,/\, \text{rtt}_i}{\text{rtt}_i * (\sum_{k \in I} \text{cwnd}_k \,/\, \text{rtt}_k)^2} * \frac{1 + \alpha_i}{2} * \frac{4 + \alpha_i}{5} \tag{2.4}$$

$$\frac{\text{cwnd}_i \,/\, \text{rtt}_i}{2} * \min(\alpha_i, 1.5) \tag{2.5}$$

Weighted Vegas (wVegas): wVegas uses a delay-based approach, and make its decisions based on packet queuing delay [36]. It aims to have a more fine-grained load balancing than LIA, OLIA and BALIA. It is based on the Vegas CCA, which is an uncoupled CCA used for TCP, with the addition of weights.

Cubic: Cubic is an uncoupled CCA [38]. It uses a cubic function instead of a linear window increase function of Standard TCP to improve utilisation in fast long-distance networks [33]. The cubic window increase function can be seen in Equation (2.6), where `C` is an aggressiveness constant, `t` is the time since the beginning of the current congestion avoidance and `W_max` is the window size just before the window is reduced in the last congestion event. `K` is calculated as per Equation 2.7, where β is the multiplication decrease factor.

$$C * (t - K)^3 + W_{max} \quad (2.6)$$

$$K = \sqrt{W_{max} * (1 - \beta)/C} \quad (2.7)$$

2.2.3 MPTCP in the Linux kernel

The MPTCP protocol is implemented on the Linux kernel by an open source project [14][18]. Development began in 2009 as an out-of-tree Linux implementation with its first periodic release v0.86 based on the v3.5.7 Linux kernel [39]. The last out-of-tree release v0.96 [18], based on the v5.4 Linux kernel, was published in February 2023. However, as of March 2020 the official Linux kernel v5.6 and onwards have included some MPTCP support, which is continuously being expanded and improved by the MPTCP coding community.

This jump from out-of-tree to in-tree has resulted in some major differences in implementation, with some functionality included in the out-of-tree v0.96 still missing in-tree [18], [19]. Table 2.7 contains a comparison between relevant functionality included in the different MPTCP version. Switching CCAs and SA:s is both possible in v0.96 with the use of modules and `sysctl` [17], [18]. However, the v5.15.60 kernel, which is the one Satcube's satellite terminal is running, has no MPTCP specific CCAs and SAs, and the `sysctl` support for SAs are missing [20]. As of the latest in-tree MPTCP the `sysctl` support for schedulers has been reintroduced along with minRatio as default scheduler and simplified Round Robin and redundant schedulers [19].

Features	**v0.96** [18]	**v5.15.60** [20]	**Latest in-tree** [19]
Cubic	✓	✓	✓
LIA	✓		
OLIA	✓		
BALIA	✓		
wVegas	✓		
sysctl CCA support	✓	✓	✓
minRatio		✓	✓
Round Robin	✓		simplified
Redundant	✓		simplified
minRTT	✓		
BLEST	✓		
ECF	✓		
sysctl SA support	✓		✓

Table 2.7: Comparison of included features in the MPTCP versions 0.96, 5.15.60 and latest.

2.2.4 Benefits and Use-cases in Satellite Communication

Benefits of using MPTCP in satellite communication include [14], [40] :

- **Reduced downtime:** Utilising several beams allows connections to move in and out of service without experiencing downtime.
- **Seamless handovers:** Move traffic from one beam to another without downtime.
- **Best network selection:** Possibility to choose the best subflow for current use-case.
- **Increased throughput:** Increased available bandwidth and throughput when combining several beams.
- **Security:** Improved security through abstraction.

MPTCP is useful in many areas of the network field. As for what is relevant for this thesis, the most apparent use-cases are when combining several satellite terminals into a meshed network. The terminals would create one WLAN on ground, but have the ability to funnel traffic through several beams. We would then be able to reap the above-mentioned benefits. A likely scenario would be if several smaller groups, each with their own satellite terminal, gathered in a larger camp. Utilising MPTCP would provide service for the entire camp on just one network. Satellite terminals could also be combined to enable heavier transmissions, such as video, rather than accommodate more users.

2.3 Related work

Previous thesis work utilised the Satcube Ku to compare how MPTCP performs over different types of links [9]. The thesis found that while throughput could be increased using MPTCP compared to normal TCP connections, the best aggregated benefits were received when the links were of the same type. Further, it concluded that larger bandwidth and longer latency of the links worsened the MPTCP performance compared to simple links. It shows that there are possible benefits of using MPTCP in satellite communications, supporting further research. The thesis's conclusions regarding the link types are what guided us to the decision to focus on GEO/GEO connections. However, the thesis does not analyse the impact of CCAs and SAs on MPTCPs performance, nor does it implement meshed and dynamic network environments.

There does exist previous research on comparisons between CCA, although they were performed in a different network environment, making the tests not very applicable to satellite communication. In [32] the delay-based coupled CCAs LIA, OLIA and BALIA are compared in terms of their CWND utilisation. They created two simulated scenarios: one with two disjoint MPTCP paths, another with the addition of a single path TCP (SPTCP) flow competing with one of the MPTCP paths. Fair bandwidth distribution was shown for disjoint path with equal RTT. However, they

conclude that in the case of heterogenous disjoint paths the CCAs cannot attain their share of bandwidth. It is also shown that when competing against SPTCP, MPTCP becomes lenient and surrenders most of the bandwidth. The problem increases as the differences in RTT grow. Their findings are relevant to this thesis, as large differences in RTT are probable when connecting to separate satellites. [32] suggests mitigating the problem using alternative methods that detect shared bottlenecks. However, these approaches are not included in the Linux kernel.

Similarly, previous research has been done comparing SAs in a different network environment. In [38] the SAs minRTT, RR, BLEST, ECF and STTF, are compared on heterogeneous paths. It concludes that the size of the intermediate network queue has significant impact on the scheduler's performance. MinRTT shows poor performance in the case of longer queues. They highlight two important metrics for measuring SA performance in goodput, i.e. transmitted data excluding headers and re-transmissions, as well as Out-of-Order (OFO) queue size. With the aim being to have high goodput and short OFO queues.

3

Experiment Design

The experiments in this thesis aim to evaluate the performance of various Scheduling Algorithms (SAs) and Congestion Control Algorithms (CCAs) within the constraints of geostationary (GEO) satellite communication networks, particularly in meshed MPTCP networks. As such the experiments are sought to explore the algorithms impact on areas in which satellite communication struggles, namely throughput, responsiveness, and jitter. Further we mean to test how the networks responds to the dynamic nature of the network through defined disturbances, such as disconnecting and adding new paths.

To test how algorithms affect those areas the test must be done using a high data load. This is as the algorithms are mainly put to work when handling much data, especially CCAs will not be put to use unless enough data is sent to fill up control windows. To make the tests easily comparable and reproducible, we will not design our own tests but use a known tool: `sockperf`, specifically the `under-load` test [41]. This test provides a high data load to the network and measurements done within the tool are done to a set amount of packets. More details on exact commands can be found under Chapter 4.

To test for synergy or dissonance between algorithms each CCA was tested against each SA, in addition to the standard algorithms implemented in the kernel at that point. The latter was done to gain a base-point to be able to see whether the algorithms improved the defaults. These tests were done iteratively in two stages: On a partially simulated test-bed, and on a live network. The tests on the partially simulated test-bed aimed for providing a ground for analysis which was entirely reproducible. The tests were done with the same end-points used in live tests. However, the network path with the transmission delay of the antenna and propagation delay of the beam were simulated. This also allowed for total control of the links, making it possible to provide equal tests for dynamic disturbances and disconnections of paths for the many combinations of algorithms. The live tests using beam connections between Satcube Ku and GEO satellites in flight above Gothenburg were done to validate the algorithms under fluctuating satellite communication conditions with true signal disruptions and environmental noise.

The following sections further define the design of our tests as well as the criteria of algorithms selection and explain the choices of collected metrics and how we expected to use them to analyse the algorithms.

3.1 Selection Criteria

The main limitations for this thesis was available time and the hardware available to test on. This is due to the limited time available when doing a thesis, and the performing of live tests. As such the selection criteria for both SAs and CCAs was mainly their ease of implementation. Further the version of the Linux kernel available on the testing hardware influenced which algorithms were easy to implement. This is in part due to the lack of existing implementations of scheduling and congestion control algorithms within the v5.15.60 kernel. Due to the rigidity of the available test-bed components the kernel version couldn't be updated.

Therefore, we chose algorithms which had been previously implemented in the Linux kernel (v0.96) which meant less work per algorithm once we had re-written some key functions to fit the new data structures present for TCP and MPTCP control in the kernel code [18]. These (and other considered algorithms) are described in Chapter 2.

Scheduling Algorithms: The five algorithms which best fit our criteria were the Round-Robin (RR), Redundant, Blocking Estimation (BLEST), Minimum Round Trip Time (minRTT), and Earliest Completion First (ECF) schedulers. Those five algorithms were the only ones implemented in the v0.96 kernel [18]. In addition, we also selected the latest default scheduler minRatio available in the up-stream version due to its high compatibility and ease of implementation in v5.15.60 [19]. We decided to use minRTT as our baseline, as it is a simple algorithm and is used as back-up in both BLEST and ECF [22], [25].

Congestion Control Algorithms: Although ease of implementation was a large contributor to the choice of CCA, they also have more easily comparable characteristics. The two main differences between the function of CCAs are whether they are coupled, and if they base the control window on losses or delays. Early on we made the choice to focus on coupled CCAs as many or most uncoupled algorithms risk breaking the fairness goal of MPTCP which was presented by the Internet Engineering Task Force with their first introduction of a MPTCP congestion control [5]. In general, it has been found that uncoupled congestion control algorithms perform better. As such it might still be interesting to expand upon this limitation given enough time.

However, continuing on the ease of implementation criteria and the existence of implemented CCA in the v0.96 kernel version, namely Linked Increase Algorithm (LIA), Opportunistic LIA (OLIA), Balanced Linked Adaptation (BALIA), and weighted Vegas (wVegas), we chose to go ahead with implementing those algorithms into the newer kernel [18]. The standard CCA implementation already present in the kernel as mentioned earlier are part of our experiments as a baseline test, hence Cubic is also tested [19].

3.2 Testcases

We designed two sets of testcases, defined below, to analyse MPTCP configurations on our two testbeds. The testcases are designed with the aim to evaluate the configurations performances in multiple environments as SA and CCA are designed to flourish in different network settings. The testbeds have different capabilities, supporting different types of testcases: The Model Testbed which simulates network links, and the Live Testbed which use actual satellite connections. Their physical setup is further described in Section 4.1.

The Model Testbed made some further testcases accessible, as the network conditions were defined directly by us. The default configuration is seen in Table 3.1. Further variances were introduced to tests in link capacity and latency. On the Live Testbed however, the network conditions were unregulated.

Configuration	**Value**
Downlink bandwidth	10 Mbps
Uplink bandwidth	3 Mbps
One-way latency	250 ms
Packet loss	1 %

Table 3.1: Default network conditions in controlled tests.

The tests were run through clients installed on the Parental Boards (PB) which are access points connecting to satellite terminals. Both testbeds includes 2 PBs, but only one is monitored for analysis due to limitations in the testing programs. All tests were run on all algorithm combinations. Our two sets of testcases are here summarized in one list, where the corresponding testbed is denoted with an *M* for Model Testbed and *L* for Live Testbed:

- **Default:** One PB sending test transmissions utilizing two paths with the same network conditions with one subflow each, while the other passively forwards transmissions.

  M.Def Both subflows using the default network conditions.

  L.Def Both subflows using the same beam.

- **Heterogenous paths:** One PB sending test transmissions on two paths with heterogenous conditions with one subflow each, while the other passively forwards transmissions.

  M.Loss One subflow with default conditions, one with an packet loss of 10 %.

  M.Latency One subflow with default conditions, one with 100 ms higher latency, i.e. a one-way latency of 350 ms.

  L.Het Subflows use different beams.

- **Simultaneous sending:** Both PBs sending on two shared paths, using two subflows per PB. One PB sending a constant stream of data utilising Cubic_MinRatio . The other PB performs test transmissions. Both passively forwards the other PBs transmissions. The aim is to evaluates the configuration's performance in a meshed multi-client environment.

  M.Sim+Def Simultaneous sending, where both paths use the default network configuration. The second PB is configured with Cubic_MinRatio for all tests.

  L.Sim+Def Simultaneous sending, where the paths use the same beam. The second PB is configured with Cubic_MinRatio for all tests.

A couple testcases were also created for TCP to create a reference for the MPTCP performance:

- **Model:** Running TCP on the Model testbed on a link with the default network configurations.
- **Live with WireGuard:** Running TCP encapsulated in a UDP tunnel on the Live testbed.
- **Live without WireGuard:** Running TCP on the Live testbed with no encapsulation.

3.3 Data Collection and Result Analysis

The analysis of the algorithms is based on collected data. This involves multiple common measurements such as RTT, throughput, and packet loss. This data was collected through the different use of tools, which is described in Subsection 4.3. The following performance metrics were measured:

Latency [ms] is the propagation delay of the network or link.

Throughput [bits/s] is the total rate of transfer of data of the network or link.

Goodput [Bytes/s] is the rate of transfer of useful data of the network or link. We favour this metric over throughput as that includes excessive data such as retransmissions, lost packets, and packet scheduler communications such as ACKs and PSHs as written about in Section 2.2.

Round Trip-Time (RTT) [s] is a useful overall measure which shows the average response time of the network or link. It becomes a quite all-encompassing value of the performance of the algorithms as it includes not only the long propagation delay caused by lengthy satellite links, but also the transmission delays caused by large queues,

Out-of-Order (OFO) queue size [Bytes] shows how good the algorithms are at sending packets in-order. This measurement is interesting to MPTCP as the long latencies of links can cause large or overflowing buffers for the OFO queue, which leads to dropped packets. Measuring the size of the OFO queue helps us separate the packets dropped due to too large control windows from those dropped from natural disturbances. It is thus mainly interesting to analyse the performance of CCAs although some SAs attempt to help alleviate OFO queue through smart use of RTT.

Flow Completion Time (FCT) [s] provides a measurement which effects are easy to relate to real-world applications. When comparing how well the algorithms handle sending burst data in different sizes, one can easily compare the real-world effects as wait time is generally the one thing that users notice.

Bandwidth utilisation [%] shows how well the scheduler combination utilises the bandwidth available to the sender.

Resource utilisation of sender shows the percentage of use of the network interface card (NIC). This was not used to judge algorithms, only to see if any metric is suppressed by any of these resources.

Results from all tests were aggregated per testcase to find the mean, max and min values for metrics per testcase and algorithm combination. To finalize this aggregation, outliers were removed. They were detected using z-score with a threshold of ± 3 and results were then recalculated when the data had been cleaned. Further, all testcases corresponding to the same testbed was aggregated to find the mean, max and min values for metrics per testbed and algorithm combination. No outliers were removed in this step, since the data had already been cleaned once.

Given the many tests multi-level aggregation was sometimes necessary leading to use of pooled standard deviation: $s_p = \sqrt{\frac{\sum_{i=1...N}(n_i-1)*s_i^2}{\sum n-N}}$ as well as pooled mean: $\frac{\sum n}{N}$. However, groups for which standard deviation is pooled should be fairly similar. Exactly how similar is quite arbitrary, for this reason we set a limit that any group must not be more than three standard deviations apart when pooling their standard deviations.

We used Cubic_MinRatio as a baseline for performance since it is the default MPTCP configuration on the 5.15 Linux kernel. Through this the performances of algorithms could be judged regarding its performance relative to the default configuration. Whenever TCP is used Cubic was used as CCA for the same reasons.

4

Implementation

4.1 Testbeds

We used two testbeds when measuring performance. These will be referred to as the Model Testbed (Figure 4.1) and the Live Testbed (Figure 4.2). They both shared some main components: Two parental boards (PBs) which emulated the satellite terminal and acted as clients, and Satcube's virtual private server Donald. Their differences lay in their network environment. In our Model Testbed each client was connected to internet via a network emulation unit (Jalapeño / TC), which in turn was connected to a shared switch which provided internet access. This testbed gave us full control over the network environment by configuring the Jalapeño, which allowed us to create clean and reproducible tests. The Live Testbed on the other hand, accessed internet via real satellites This allowed us to run tests in a more realistic although more unreliable environment. In the Live Testbed each client was connected to a satellite terminal, accessing its antenna which connects it to internet via a satellite beam.

Both testbed configurations allowed for two subflows per PB, one subflow directly to internet via Ethernet port 2, and another through the other PB via Ethernet port 3. Additionally, we had a third testbed, called our Debug Testbed, which was used in the initial stages when our algorithms where implemented and tested. Its setup is similar to the Model Testbed, with the exception that only one PB was used which then connected to two Jalapeños, allowing for two subflows.

4.1.1 Parental Board (PB)

A satellite terminal, specifically a Satcube Ku, was represented by a System-on-Module (SoM) mounted on a PCB, provided by Satcube, hereafter referred to as the parental board (PB). The SoM had the following specifications as seen in Table 4.1:

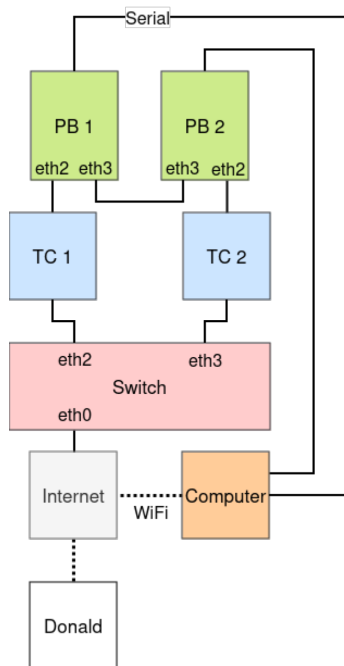

Figure 4.1: Overview of the Model testbed.

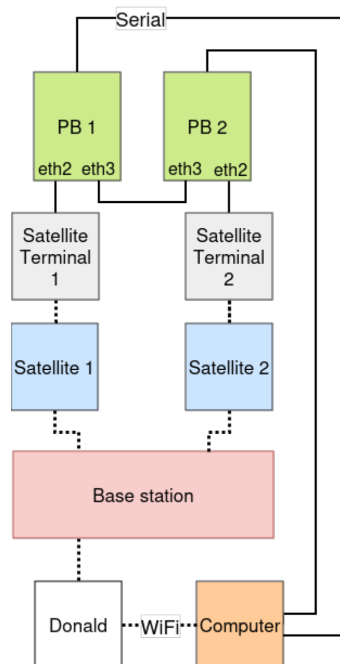

Figure 4.2: Overview of the Live testbed.

System		Parental Board with SoM
Operating System (OS)		Yocto Kirkstone (5.15.60)
Processor	Clock Speed Cores	1600 MHz Quad
Memory	Type Size	LP-DDR4 1024 MB
Storage	Type Size	eMMC 8 GB
Interfaces	Ethernet USB	4x -

Table 4.1: Technical details of Parental Board with its mounted System-on-Module.

The four Ethernet ports of the PB was used when forming the meshed network Additionally, an external micro-USB port existed for serial communication through which commands could be run directly on the SoM mounted on the PB.

The PBs IP information was configured on the board using NetworkManager [42] via the `nmcli` command. An example of the setup command can be seen in Listing 1. The settings for the Ethernet port connected to internet was set such that it is on the same subnet as its corresponding Ethernet port on the network switch. On the Model and Live Testbed the Ethernet ports connecting the two PBs had its own subnet, which is visualised in Figure 4.3. To enable forwarding of in- and outgoing traffic via the other PB both had to be configured using `sysctl` and `iptables` as seen in Listing 2.

```
PB:/$ nmcli c modify eth2  connection.autoconnect yes \
ipv4.method manual \
ipv4.addresses 192.168.10.2/24 \
ipv4.gateway 192.168.10.1 \
ipv6.method ignore
```

Listing 1: Example of configuring relevant NetworkManager settings for eth2 on the Parental Board.

```
PB:/$ sysctl -w net.ipv4.ip_forward=1
PB:/$ iptables -A FORWARD -i eth3 -o eth2 -j ACCEPT
PB:/$ iptables -A FORWARD -i eth2 -o eth3 -m state \
--state ESTABLISHED,RELATED -j ACCEPT
PB:/$ iptables -t nat -A POSTROUTING -o eth2 -j MASQUERADE
```

Listing 2: Configurations to enable forwarding on in- and outgoing traffic between Parental Boards on the Model and Live Testbed.

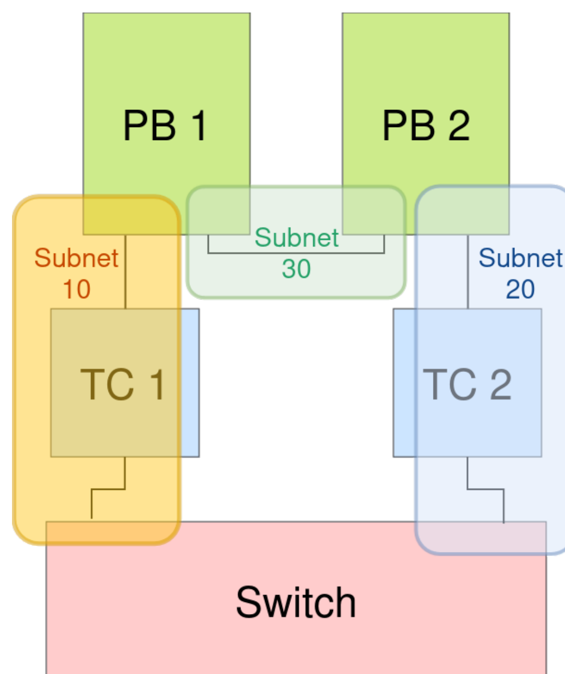

Figure 4.3: Visual representation of the implemented subnet separation on the testbeds.

4.1.2 Network Switch

To connect the Model Testbed to internet a network switch with 5 Ethernet ports was used. We configured the switch with two VLAN:s:`VLAN10` and `VLAN20`. The VLAN:s allow for two separate subnets for the two connections between the PBs and

the switch. `eth2` and `eth3` on the switch has Port VLAN ID VLAN10 and VLAN20 respectively. The switch's internal IP can be accessed via the `192.168.1.x` subnet on non-VLAN ports. An overview of the configurations for the ports on the switch can be seen in Table 4.2.

Ethernet port	Configuration	Connected to
0	Default DHCP	Internet
1	Default	Computer
2	On VLAN10	PB via Jalapeño
3	On VLAN20	PB via Jalapeño
4	Default	Nothing

Table 4.2: Configuration of network switch.

4.1.3 Network Emulation (Jalapeño)

Jalapeño boards developed by 8devices was used for emulating network conditions such as delay, disturbances and congestion [43]. Its specifications can be seen in Table 4.3. The device itself was configured to be invisible to the network, and only echoed information after applying emulated network conditions onto the data. To stay consistent, Ethernet port 0 was always connected to the PB and Ethernet port 1 to the switch. Set-up was done through the micro-USB UART port connected to a computer and the `tc` command [44]. An example of increasing latency and limiting outgoing traffic can be seen in Listing 3. A `tc` command has the following structure: `tc [options] qdisk [action] dev DEV [parent id | root] [handle id] [qdisc specific parameters ]`. The "handle" is the major number of a qdisk child.

Relevant `tc` options:

- `del dev DEV root` remove classless qdisk
- `replace`: If node does not exist, create it.
- `tbf`: Token Bucket Filter. Can slow traffic down to a precisely configured rate.
- `netem`: Network Emulator. Can add delay, packet loss, duplication etc. to outgoing packets.

```
Jalapeño:/$ tc qdisc add dev eth0 root handle 1: \
tbf rate 3mbit burst 1540 latency 100ms
Jalapeño:/$ tc qdisc add dev eth0 parent 1:1 handle 10: \
netem delay 250ms
```

Listing 3: Example of limiting bandwidth and forcing latency for outgoing traffic on eth0 using the `tc` command.

System		Jalapeño
Operating System (OS)		OpenWRT (Linux)
Processor	Clock Speed	700 MHz
	Cores	Quad
Memory	Type	NAND
	Size	128 MB
Storage	Type	DDR3
	Size	256 MB
Interfaces	Ethernet	2x1000 Base-T
	USB	A 2.0, A 3.0, B (UART)

Table 4.3: Technical details of network emulator board (Jalapeño).

4.2 MPTCP Setup

MPTCP is part of the Linux kernel on which Satcube had built changes upon. It was already set up with correct build configurations required to use MPTCP:

```
CONFIG_MPTCP=y
CONFIG_IPV6=y
CONFIG_INET_MPTCP_DIAG=y
```

It had also already enabled the creation of MPTCP sockets by having the MPTCP `sysctl` option `net.mptcp.enabled` set to 1 [45].

4.2.1 Subflows and Routing

To force the creation of MPTCP sockets instead of TCP, `mptcpize` was used [46]. This was done by wrapping the program with the mptcpize command:

```
mptcpize run <program> [program arguments]
```

The MPTCP routing was configured manually using `ip mptcp` together with `ip route` and `ip rule`. A simplified shell script setting up our routing information for one PB in the dual PB testbed can be seen in Listing 4. When two PBs were used they were configured to forward the other's passing transmissions. This was done by having the ipv4 `sysctl` option `net.ipv4.ip_forwarding` set to 1 as well as configuring `iptables` for forwarding between the relevant interfaces.

4.2.2 Congestion and Scheduler Configuration

When the system was initialised the kernel used Cubic as default CCA, with Reno available as an alternative, and minRTT (implemented by us) as default SA. All others were implemented as modules, hence not being compiled directly into the kernel. These instead had to be loaded at runtime using `modprobe`. It was then possible to set them using `sysctl`. An example of this workflow can be seen in Listing 5.

```
# Setting up subflows
ip mptcp limits set subflow 2 add_addr_accepted 2
ip mptcp endpoint add 192.168.10.2 dev eth2 subflow
ip mptcp endpoint add 192.168.30.1 dev eth3 subflow

# Create 2 routing tables
ip rule add from 192.168.10.2 table 1
ip rule add from 192.168.30.1 table 2

# Configure routing tables
ip route add 192.168.10.0/24 dev eth2 scope link table 1
ip route add default via 192.168.10.1 dev eth2 table 1
ip route add 192.168.30.0/24 dev eth3 scope link table 2
ip route add 192.168.30.0/24 via 192.168.30.2 dev eth3 table 2

# Default route for the selection process of normal internet traffic
ip route add default scope global nexthop via 192.168.10.1 dev eth2
```

Listing 4: Simplified shell script setting up MPTCP subflows and routing using the ip command.

```
PB:/$ sudo sysctl net.ipv4.tcp_congestion_control
net.ipv4.tcp_congestion_control = cubic
PB:/$ sudo sysctl net.mptcp.scheduler
net.mptcp.scheduler = minRatio
PB:/$ sudo modprobe mptcp_lia
PB:/$ sudo modprobe mptcp_ecf
PB:/$ sudo sysctl -w net.ipv4.tcp_congestion_control=lia
net.ipv4.tcp_congestion_control = lia
PB:/$ sudo sysctl -w net.mptcp.scheduler=ecf
net.mptcp.scheduler = ecf
```

Listing 5: Example of loading and setting Congestion Control Algorithm LIA and Scheduling Algorithm ECF during runtime using sysctl.

4.2.3 Encapsulating packets on Live Testbed

When sending data on the Live Testbed we utilize WireGuard (WG) [47] to prevent the TCP headers from being stripped by the Modem installed on Satcube Ku. WG is a VPN tunnel which encapsulated each MPTCP packet, hence protecting the header. The decision to use WG was informed by the use of WG in a previous thesis

at Satcube [9], which also used a Satcube Ku.

The WG server is set up on our server and two clients, one for each subflow, is set up on one of the PBs. The clients are indirectly connected to a interface through packet routing configurations using `fwmarks` and `ip route`. Each WG client is assigned its unique fwmark and each fwmark connects to its own routing table. One routing table routes to Ethernet port 2, while the other to port 3. At the client side, all packets are marked with its WG client fwmark, ensuring the right interface is used. The MPTCP endpoints are then set to the WG clients, instead of the Ethernet ports directly.

4.3 Performance Evaluation Tools

In order to review performance of the network multiple tools were used to generate data and monitor different system characteristics. `Sockperf` [41] and `iPerf3` [48] was used to generate data as well as monitor simple metrics. For deeper analysis Wireshark [49] and Sar [50] was used. Each tool is explained in further detail in its respective subsection below. The Key Performance Indicators (KPI) which we will mainly focus on, and the tools used for collecting them, can be seen in table 4.4.

KPI	**Tool**
Round-Trip Time (RTT)	Sockperf
Latency	Sockperf
Throughput	iPerf3
Goodput	Wireshark / tshark
Flow Completion Time (FCT)	Wireshark / tshark
Send Window	Wireshark / tshark
Retransmissions	Wireshark / tshark
Out-of-Order buffer	Wireshark / tshark
Network Interface Card (NIC) utilisation	sar

Table 4.4: Overview of Key Performance Indicators and tools used.

4.3.1 iPerf3

iPerf3 [48] is a tool for active performance measurement on IP networks. It generates TCP traffic and measures throughput, latency, missed packets and CWND for the entire connection and reports a summary of each metric in discrete time frames of 1 second each [51]. A test can be tuned through options where the following are a relevant subset:

- `-s` Run in server mode
- `-c <host_ip>` Run in client mode, connecting to iPerf server on <host_ip>
- `-p <port>` The port to connect/listen to. Default is 5201

- `-t <number>` The time in seconds to transmit for. Default is 10 seconds.
- `-R` Run in reverse mode (server sends, client receives)
- `-logfile <filename>` Send output to a log file
- `-J` output in JSON format

Further, the use of MPTCP sockets can be enforced by wrapping the iPerf3 command with `mptcpize` on both the server and client. This is an example of running iPerf3 on the server and client respectively:

- **Server (Donald)**: `mptcpize run iperf3 -s -p 5001`
- **Client (PB)**: `mptcpize run iperf3 -c <donald_ip> -p 5001 -t 30 -logfile <filename>`

iPerf3 was used to measure throughput, ..., and .... All iPerf3 tests was 30 seconds long and the value from each test was extracted from iPerf3's own summary of each test. On the both testbeds we ran 10 tests per testcase, resulting in 40 and 30 tests per algorithm combination on the Model and Live testbed respectively.

4.3.2 Sockperf

Sockperf [41] is a tool for active measurement of throughput and latency which allows for set-up through single point and use of configuration files. The tests are performed through generation of UDP or TCP traffic and monitoring of the packets, much like iPerf3. However, it has predefined testing modes such as `ping-pong` and `under-load` which improves ease of testing. Sockperf also provides per packet latency at high resolution rather than a latency-per-second average like iPerf3. The following are some relevant Sockperf options:

- `sr` Run sockperf as server.
- `-tcp` Use TCP protocol. Default is UDP.
- `-i <ip>` Listen on/send to ip <ip>.
- `-tcp-avoid-nodelay` Stop/Start delivering TCP Messages Immediately.
- `-full-log <filename>` Dump full log of all messages send/receive time to the given file in CSV format.
- `ul` Run sockperf client for latency under-load test.

Besides the full log, Sockperf also prints a summary to standard output. This can be re-directed to a file with the `&>` command. Finally, to enforce the use of MPTCP sockets the sockperf command can be wrapped with `mptcpize` on both the server and client. The following is an example of running a `under-load` Sockperf test on the server and client respectively:

- **Server (Donald)**: `mptcpize run sockperf sr -i <donald_ip> -tcp`
- **Client (PB)**: `mptcpize run sockperf ul -i <donald_ip> -tcp -tcp-avoid-nodelay -full-log <filename> &> <filename2>`

Sockperf were used to measure latency. All our `sockperf` tests were 30 seconds long under-load tests. For each test sockperf provides a *summary latency* value which we used as the result for a test. On both testbeds 20 tests were run for each testcase.

This resulted in 80 tests on the Model Testbed, and 60 on the Live Testbed, per algorithm combination. 20 tests was also run for each of the TCP testcases.

4.3.3 Sar

Sar (System Activity Report) [50] is run on the client and measures the resource utilisation. This includes processor statistics such as CPU and RAM utilisation in addition to network statistics such as packets and bytes transmitted/received. The monitoring is configured through Sar's options where these are some relevant ones:

- `-u` Monitor CPU
- `-r` Monitor RAM
- `-n DEV` Monitor network devices
- `-iface=<interfaces>` Specify the wanted network interfaces
- `<duration>` Collect data for a total of <duration> seconds

The following is an example of monitoring Ethernet port 2 and 3 for 40 seconds with Sar as a background process, allowing you to run iPerf3 or Sockperf on top:

```
sar -n DEV -iface=eth2,eth3 1 40 > <filename> &
```

Sar was used to inspect the distribution of packets on the subflows to verify expected behaviour of the scheduling algorithms. A sar test was run in the background of all `iPerf3` and `sockperf` under-load tests. It was set to 35 s, since all `iPerf3` and `sockperf` tests where 30 s long, to allow for eventual application delays.

4.3.4 Wireshark

Wireshark [49] is a packet analyser. It provides much of the same measurements as `iPerf3` and `Sockperf` with the addition of an UI, protocol details for every packet as well as the ability to generate graphs for most performance indicators. To create additional specialised graphs we have processed the Wireshark data with our own Lua [52] scripts. The following command can be run locally to remotely capture packets and analyse them in a local Wireshark process:

```
ssh <user@donald_ip> 'echo <sudo_psw> | sudo -S tcpdump -U -i eth0 -w -'
| wireshark -i - -k
```

To break down the command: an ssh connection is established to our server and the command within the single quotes is executed remotely. `tcpdump` [53] is used to capture the package data on Ethernet port 0 on the server, which is then written to standard output because of the `-w -` option. This in turn prints the data on our local standard input. Wireshark runs locally and starts capturing immediately because of the `-k` flag. Because of the `-i -` option, it captures the data from the standard input.

FCT, goodput, send window size, retransmissions and OFO buffer was measured with Wireshark during the run of `iPerf3` tests. Wireshark captured packets during five iPerf3 tests of each testcase, resulting in 20 tests for the Model Tested and 15 for the Live Testbed.

TShark is a terminal based version of Wireshark which without any given arguments functions the same as tcpdump. However, we utilised it for command line parsing of `.packg` files with Wireshark's dissectors, including our custom post-dissectors written in Lua and column layouts.

4.4 Algorithm Details

All CCAs and SAs we used, except for Cubic, were implemented on the v5.15.60 kernel by us using a combination of v0.96 code and the latest in-tree MPTCP version. This section provides further details on our implementation strategies.

4.4.1 Congestion Control Algorithm Implementation

We implemented LIA, OLIA, BALIA and wVegas. Code for all of them can be found in the v0.96 MPTCP release [18] but is not included in the 5.15.60 Linux kernel [20]. We imitated the old implementations as closely as possible. The old MPTCP CCA code fit well with the v5.15.60 kernel and required only slight modifications. Modifications included updating functions names and rewriting small sections of code where the functionality had been removed from the header files. The functionality for setting the CCA using `sysctl` was already in place since TCP also use it. We simply had to implement our CCAs as TCP congestion control modules, making it possible to load them using the same `sysctl` command.

4.4.2 Scheduling Algorithm Implementation

We implemented minRatio, RR, ECF, minRTT and BLEST as modules into the Linux kernel, except for minRTT which is always compiled in the kernel and initialised as the default scheduler. No other scheduler modules exist in v5.15.60 [20]. It was decided to exclude redundant scheduling to respect our time constraint since it would require a more extensive rewrite of the code to adapt the message sending process to accommodate for redundancy. Our implementations are a mix of v0.96 and the latest in-tree code with necessary adaptations made to ensure compatibility with v5.15.60. A lot off relevant code has gone through major refactoring between v0.96 and v5.15.60, so simply copying the old code to our kernel wasn't working. Further implementation details for each scheduler can be found in its respective paragraph below.

The functionality for setting SAs using `sysctl` exists in the v0.96 as well as recent kernel releases, but not in the v5.15.60 kernel and thus had to be re-implemented by us. We ended up using the latest in-tree MPTCP code since it was more compatible than the old code and implement the `sysctl` command `mptcp.scheduler` in our kernel.

minRTT: A simple scheduler that picks the available subflow with the minimum RTT. minRTT is implemented as the default scheduler in v0.96[18]. Since it is our

baseline scheduler, this is also the case in our implementation, hence it is not a module. Pseudocode for our implementation can be seen in Algorithm 1. Our algorithm and the reference have some major differences when it comes to the handling of retransmissions. Traditionally, minRTT will send retransmissions on the first available subflow, as long as it hasn't been sent on that subflow before. The v5.15.60 code does not have an intuitive way to keep track of what subflows the message has been sent on previously. To avoid extensive rewrites of the code, we instead borrowed the retransmission logic used in minRatio and applied for minRTT as well.

Algorithm 1 minRTT scheduler. Adapted from code in [18].

```
selected ← NULL
if msg is retransmission then
    selected ← get_active_subflow_empty_rtx_write_queue()
    if selected then
        return selected → subsocket
    end if
else                                        ▷ Send on subflow with min RTT
    min_rtt ← ∞
    for each subflow do
        if subflow is active then
            if rtt_subflow < min_rtt then
                selected ← subflow
                min_rtt ← rtt_subflow
            end if
        end if
    end for
    if selected then
        return selected → subsocket
    end if
end if
return NULL                                 ▷ No available subflows
```

minRatio: The default scheduler for MPTCP in the v5.15.60 Linux kernel, as well as in the current development version of the upstream MPTCP Linux Kernel [19], [20]. It selects the active subflow with the minimum occupied window space to total window space ratio. Our implementation is a close copy of it with some minor adjustments to make it into a loadable module. Pseudocode of our implementation of minRatio can be seen in Algorithm 2.

Round-Robin (RR): This schedules packets on subflows in a round-robin fashion [17]. The scheduler iterates the list of subflow, starting from the most recently used, and picks the next available subflow. Our implementation is a combination of code from the RR module in v0.96 and the simplified version in the latest MPTCP in-tree. Pseudocode of our implementation can be seen in Algorithm 3.

Algorithm 2 minRatio scheduler. Adapted from code in [19].

```
selected ← NULL
if msg is retransmission then
    selected ← get_active_subflow_empty_rtx_write_queue()
    if selected then
        return selected → subsocket
    end if
else
    min_ratio ← ∞
    for each subflow do
        if subflow is active then
            ratio =(queued in subflow wnd) / (total subflow wnd space)
            if ratio < min_ratio then
                selected ← subflow
                min_ratio ← ratio
            end if
        end if
    end for
    if selected then
        return selected → subsocket
    end if
end if
return NULL                                         ▷ No available subflows
```

Algorithm 3 Round-robin scheduler. Adapted from code in [18], [19].

```
r ← recently_used_subflow
n ← (r → next)
while n ≠ r do
    if n is available then
        return n → subsocket
    end if
    n ← (n → next)
end while
if r is available then
    return r → subsocket
end if
return NULL                                         ▷ No available subflows
```

Blocking Estimation (BLEST): Its scheduling decisions are based on the MPTCP's send window and the available subflows RTT estimates, MSS and congestion windows. It begins by finding the fastest subflow (lowest RTT), independent of its availability. If it is available it will use its subsocket, if not it will prompt the minRTT scheduler for the fastest available subflow, henceforth called "the slower subflow". It will then make a series of calculations to decide if it should schedule on the slower subflow or wait for the fastest subflow to become available. It uses the

correction factor λ to improve its decisions, increasing lambda when blocking occurs and decreasing it when blocking has been avoided. Our implementation is inspired by code from the BLEST module in v0.96 and pseudocode of our implementation can be seen in Algorithm 4.

Algorithm 4 BLEST scheduler. Adapted from [18], [25].

```
F ← get_fastest_active_subflow()
if F is available then
    return F → subsocket
end if
S ←minRTT()                                        ▷ Get fastest available subflow
rtts ← srtt_S/srtt_F
λ ← update_lambda()
X ← MSS_F ∗ (cwnd_F + (rtts − 1)/2) ∗ rtts
if X ∗ λ ≤ [MPTCP window] − MSS_S ∗ (inflight_S + 1) then
    return S → subsocket
end if
return NULL                                        ▷ wait for F
```

Earliest Completion First (ECF): Similar to BLEST in the way that it compares a fast unavailable subflow to a slower available one [22]. However, it uses other metrics to determine if it should wait for the fast subflow or use the slow. If the fast subflow is also available, it will immediately schedule on it. If not, it will use the two subflows CWND:s and RTT:s as well as the data to send to calculate the arrival time of the data if it where to be sent on the slow or fast subflow respectively [22], [28]. If the arrival time of the faster subflow is significantly earlier it will wait for it to become available. Our implementation is inspired by code from the ECF module in v0.96 and pseudocode of our implementation can be seen in Algorithm 5.

Algorithm 5 ECF scheduler. Adapted from [18], [38].

```
F ← get_fastest_active_subflow()
if F is available then
    return F → subsocket
end if
S ←minRTT()                                          ▷ Get fastest available subflow
δ ← max(σ_F, σ_S)                                    ▷ σ denotes the RTT variation
sndbuf ← data length to send
x_F ← max(sndbuf, cwnd_F * mss)
time_F ← srtt_F * (x_F + cwnd_F * mss)
time_S ← (srtt_S + δ) * cwnd_F * mss
if β * time_F < β * time_S + waiting * time_S then   ▷ waiting is 0 or 1
    x_S ← max(sndbuf, cwnd_S * mss)
    if x_S * srtt_S ≥ cwnd_S * mss * (2 * srtt_F + δ) then
        waiting ← 1
        return NULL                                  ▷ wait for F
    end if
else
    waiting ← 0
end if
return S → subsocket
```

5

Results

This chapter presents the results from the tests described in Section 3.2. The showcased results are plotted utilising the python library Matplotlib alongside Numpy for standard deviations and mean calculations. This was used alongside own additions to calculate z-score and remove outliers based on it as described in 3.3 as well as functions for pooled standard deviations and means as described in the same section.

Outliers were processed for the latency results acquired from the Sockperf under-load tests and the throughput results acquired from the iPerf3 logs. The latency results were then stripped from outliers. The throughput results did however not contain any. All Wireshark data was within the limits of standard deviations in order to aggregate, so no tests were excluded. Some failed tests were detected when processing the data and was excluded from our results:

- **L.Def** Sockperf under-load test 7 - LIA ECF
- **L.Mod+Def** Sockperf under-load test 3 - OLIA BLEST

Many graphs will show the algorithms grouped mainly by Congestion Control Algorithm (CCA), since when compiling the data the CCAs yielded quite conclusive results. Scheduling Algorithms (SAs) on the other hand did not clearly show any consistent improvements. The different test cases yielded quite similar data, as such they were aggregated together in many graphs. As mentioned in Section 3.3 we will use Cubic_MinRatio , the current default algorithms used in MPTCP, as the baseline for performance. This is highlighted in many graphs by a dotted gray line. The Live testbed is the main focus for our results as they are the most representative of real-world applications even with our tests being encapsulated in WireGuard data due to the Modem limitations. Differences noticed between the Live and Model tests are discussed further in Section 5.4 and in the Conclusion.

5.1 Reference performance of testbeds

Sockperf under-load tests using TCP, with SAR in the background, were performed to act as reference to our MPTCP results, as mentioned in Section 3.2. The results are aggregated to mean values for each testcase and can be found in Table 5.1.

TCP test	Mean Latency [ms]	Mean Throughput [Mbps]
Model testbed	1934	0.57
Live testbed with WireGuard	1775	0.57
Live testbed without WireGuard	654	1.05

Table 5.1: Mean results for TCP using Cubic from Sockperf under-load results. Latency is computed from Sockperf's logs while throughput the transmitted kB/s measured by SAR.

An initial observation is that the performance is significantly improved when not using WireGuard for UDP tunnelling on the Live Testbed. One explanation for the large impact of WireGuard on latency specifically may be because Sockperf measures latency by dividing the Round-Trip Time by two, which includes the time it takes to encapsulate and decapsulate the encryption [41]. It is also worth noting that the performance on the Model testbed, which does not use WireGuard, is just as poor as on the Live testbed when using WireGuard. This indicates that the Model testbed does not accurately emulate the SATCOM network used on the Live testbed. Although, the modelled 1% is inline with the percentage of packet loss usually modelled, as exemplified by [3], it could be poorly formatted in being too consistent throughout time, rather than concentrated around some timeslots such as might be expected when disturbances are caused by physical obstructions. This may force a lower CWND, resulting in low throughput. It is also possible that the Jalapeño can not handle enforcing the set latency without running out of buffer space. These two hypotheses are further investigated when comparing the result for MPTCP between the Model and Live testbed in Section 5.4. Due to the uncertainty regarding the Model testbeds performance, the results on Live testbed are prioritised moving forward.

Overall, the performance of all TCP tests are worse than anticipated. Much lower latencies were expected, since the propagation delay set in the Model testbed was 250 ms and the propagation delay of the beam measured to be about 300 ms. As for throughput, much higher values were expected since the uplink and downlink on the Model testbed was set to 3 and 10 Mbps respectively, and similar bandwidth was expected on the Live testbed. One possible reason for poor performance on the Live Testbed could of course be because there may exist other traffic on the network. Another may be the characteristics of Sockperf, which affect both Model and Live tests. As previously mentioned, Sockperf's latency measurements include processing time, which inflates the result. As for throughput, the Sockperf under-load tests are not technically designed to test throughput; hence, they may not structure data and send patterns to reach maximal throughput.

Comparisons of the reference data to the MPTCP results is done in the following Sections. However, as throughput results was computed from iPerf3 tests, rather than Sockperf, they can not be compared one-to-one. For easier comparison of

throughput between reference and MPTCP, the Sockperf under-load test and SAR measurements was aggregated in the same fashion for some of the MPTCP testcases. To be as close to the nature of the TCP tests, only results from testcases M.Def and L.Def using Cubic was used. The results are presented in Table 5.2. The expectation is that MPTCPs throughput should be roughly 200% to that of TCP, since it is able to utilise two paths rather than one. The results show that MPTCP was able to attain 156% of TCPs throughput on the Model testbed and 212% on the Live, which is satisfactory considering that Sockperf under-load is not optimised to achieve maximum throughput.

MPTCP Testcase	**SA used with Cubic**	**Mean Throughput [Mbps]**
M.Def	MinRatio	0.70
	MinRTT	0.94
	RR	0.90
	BLEST	0.93
	ECF	0.98
	Aggregation	**0.89**
L.Def	MinRatio	1.14
	MinRTT	1.17
	RR	1.26
	BLEST	1.24
	ECF	1.23
	Aggregation	**1.21**

Table 5.2: Mean throughput for MPTCP using Cubic from SAR measurements during Sockperf under-load results.

5.2 Performance Comparison of Congestion Control Algorithms

When reviewing the throughput on the Live testbed it is clear that the used CCA has a larger impact on performance than the SA, as seen in Figure 5.1 where the SAs overall perform roughly the same. Focusing instead on the CCAs, Cubic is shown to be overall superior to all other CCAs. This was expected due to Cubic being the only uncoupled CCA, allowing it to acquire more bandwidth than others by sacrificing fairness to single path TCP subflows. The goals can be read about in Section 2.2.2, and were defined in tandem with the release of LIA. Cubic, combined with any SA, outperformed all other combinations as seen in Figure 5.2. The Figure also interestingly show that the default configuration, Cubic_MinRatio , was best out of all combinations in terms of throughput, reaching just above 1.5 Mbps. wVegas on the other hand has considerably bad performance in terms of throughput and are by far the worst out of all the CCAs and in combination with any SA. LIA, BALIA and OLIA all have very similar overall throughput, as seen in Figure 5.1. However, given that a single combination is chosen when configuring MPTCP it is still significant to mention that out of the three, BALIA_MinRTT delivered the highest throughput,

seen in Figure 5.2.

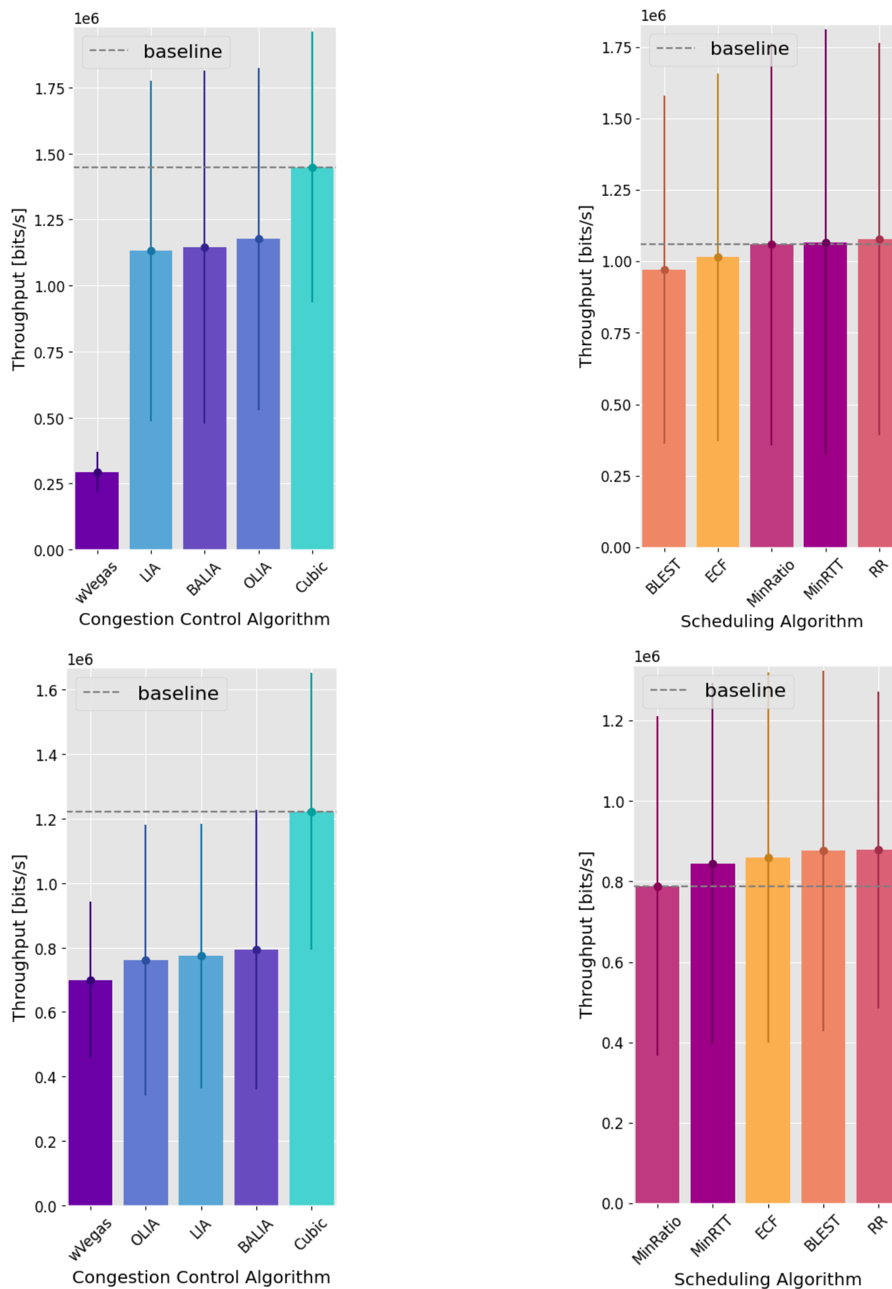

Figure 5.1: Mean throughput per CCA (left) and per SA (right), with standard deviations, from aggregation of iPerf3 results on the Live Testbed (top) and Model Testbed (bottom). The baseline marks the mean throughput of Cubic and MinRatio respectively.

Continuing with the in the on the Live testbed, but switching focus to the latencies, Figure 5.3 and 5.4 shows that wVegas was consistently best reaching the lowest latencies at just under 1500 ms. The consistently low latencies which wVegas achieved ties into its low throughput, as looking at the amount of retransmitted packets in

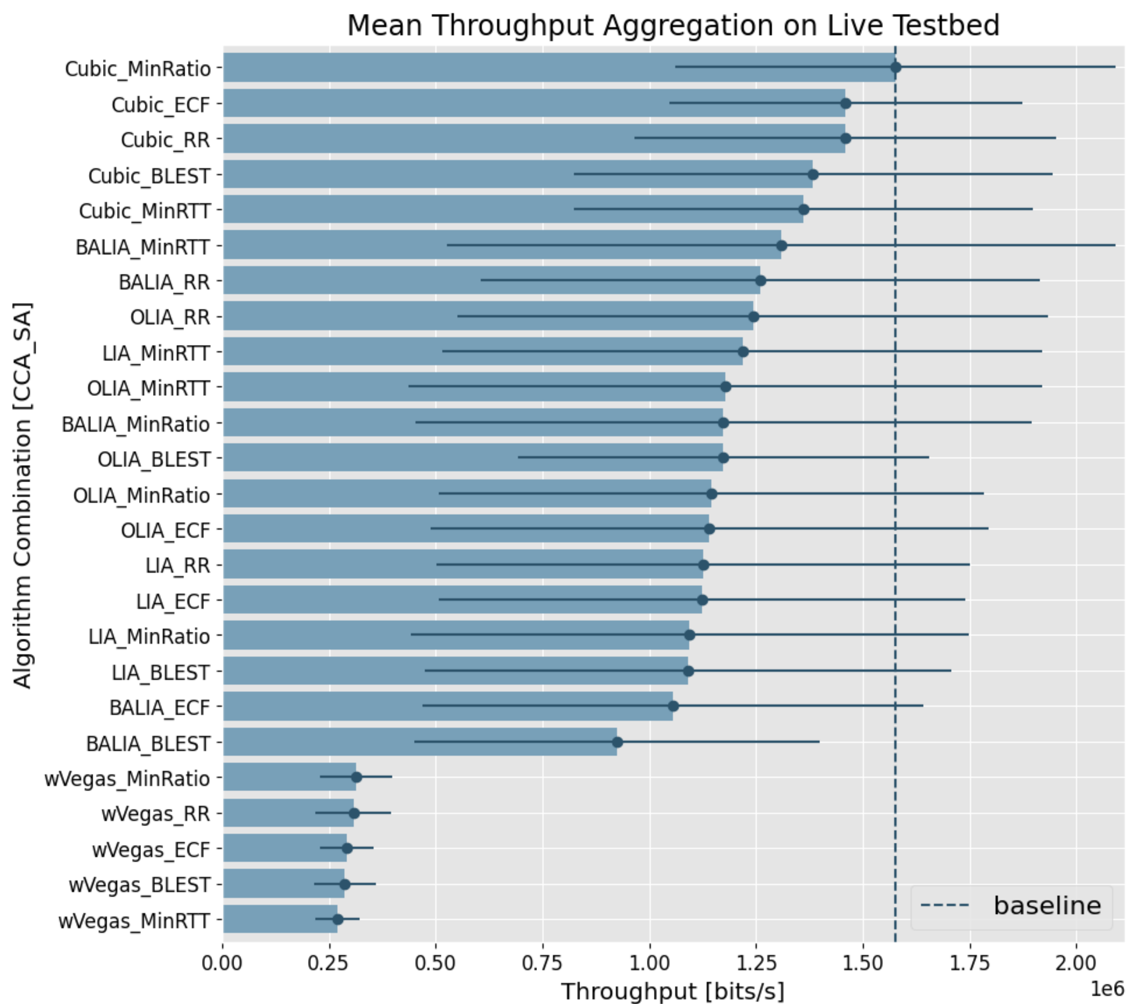

Figure 5.2: Mean throughput, with standard deviation, from aggregation of iPerf3 results on the Live Testbed. The baseline marks the mean throughput of Cubic_MinRatio .

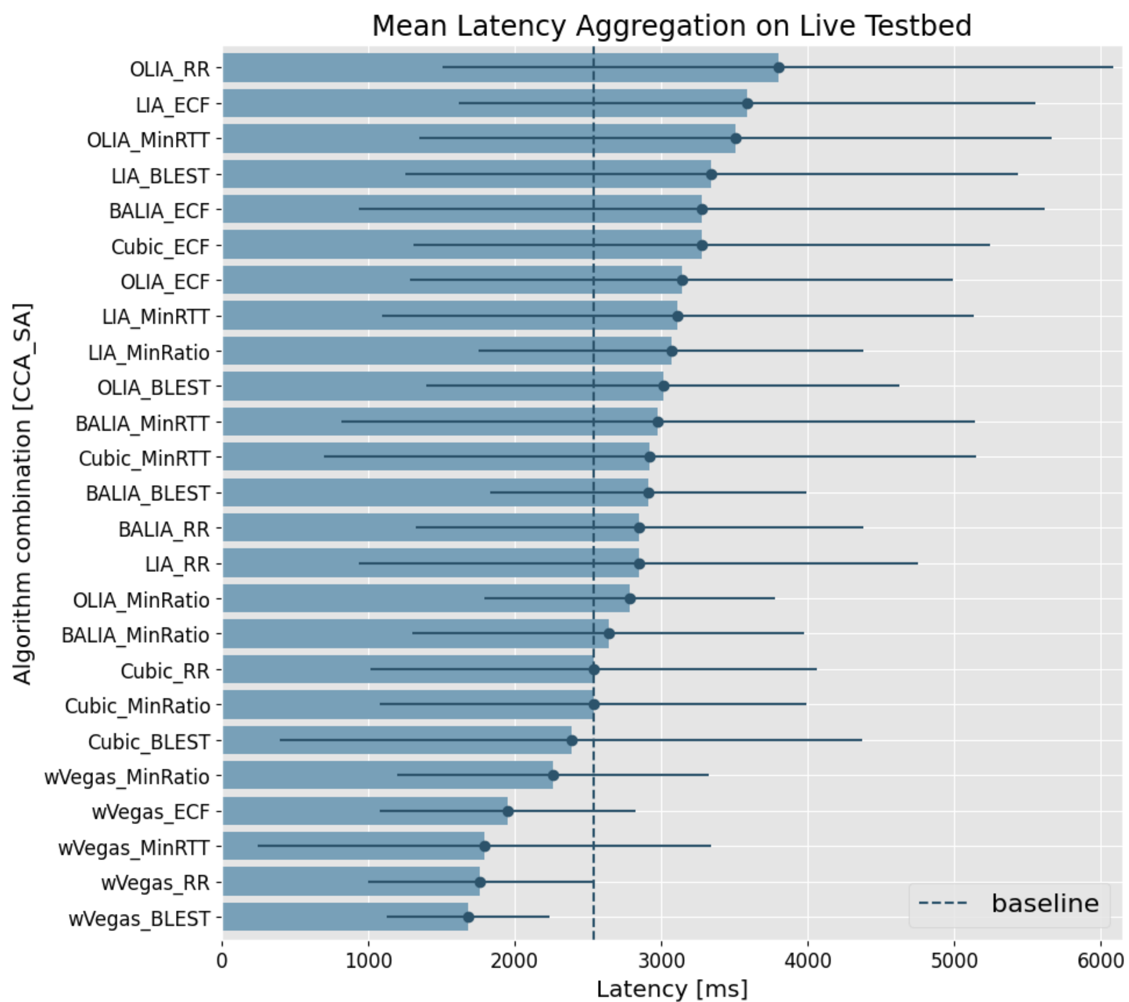

Figure 5.3: Mean latency, with standard deviations, from aggregation of Sockperf under-load results on the Live Testbed. The baseline marks the mean latency of Cubic_MinRatio .

Figure 5.11 shows that all combinations using wVegas experiences very few retransmissions. However, as visualised in Figure 5.6 the low throughput the CCA achieved is mirrored in a low goodput, clearly showing that the low throughput is not a direct result of only sending less scrap data, but also due to not managing to send a lot of the data at all. If anything this shows that while wVegas might not be a good performer overall, it could still be a good choice for time sensitive use-cases.

Further observing the aggregated throughput split into CCAs and SAs, Figure 5.1 shows that at large the CCA algorithms perform comparatively better on the Live tests compared to the Model, with Cubic being the overall top-performer on both the Live and Model tests. Similar relationships are seen for aggregating latency across different test cases and opposing algorithms as seen in Figure 5.4. Both charts also shows performance per SA in which MinRatio is outperformed, with BLEST achieving slightly lower latencies on the Live tests, and all but ECF achieving lower latencies on the Model testbed. This can be seen better in Figure 5.2 where MinRatio

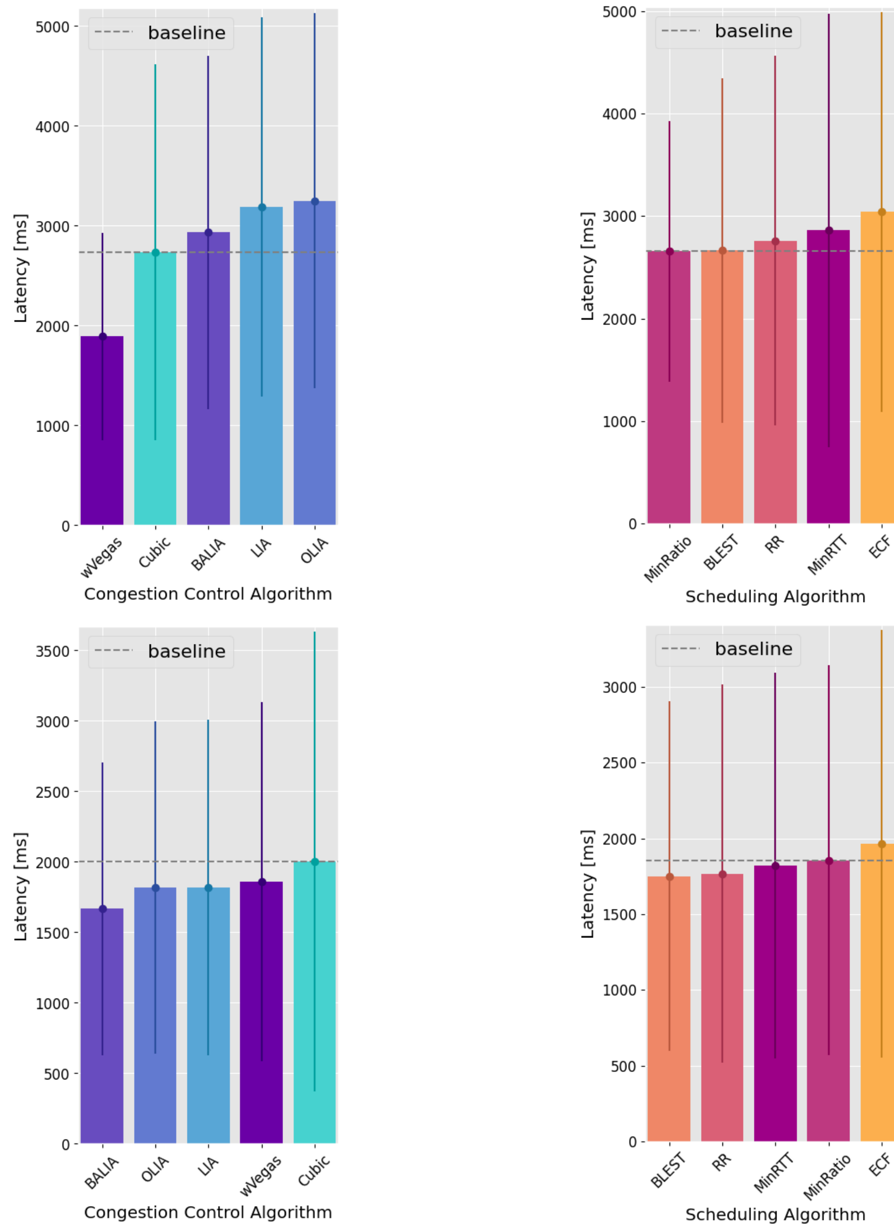

Figure 5.4: Average latency per CCA (left) and per SA (right), with standard deviations, from aggregation of Sockperf under-load results on the Live Testbed (top) and Model Testbed (bottom). The baseline marks the average latency of Cubic and MinRTT respectively.

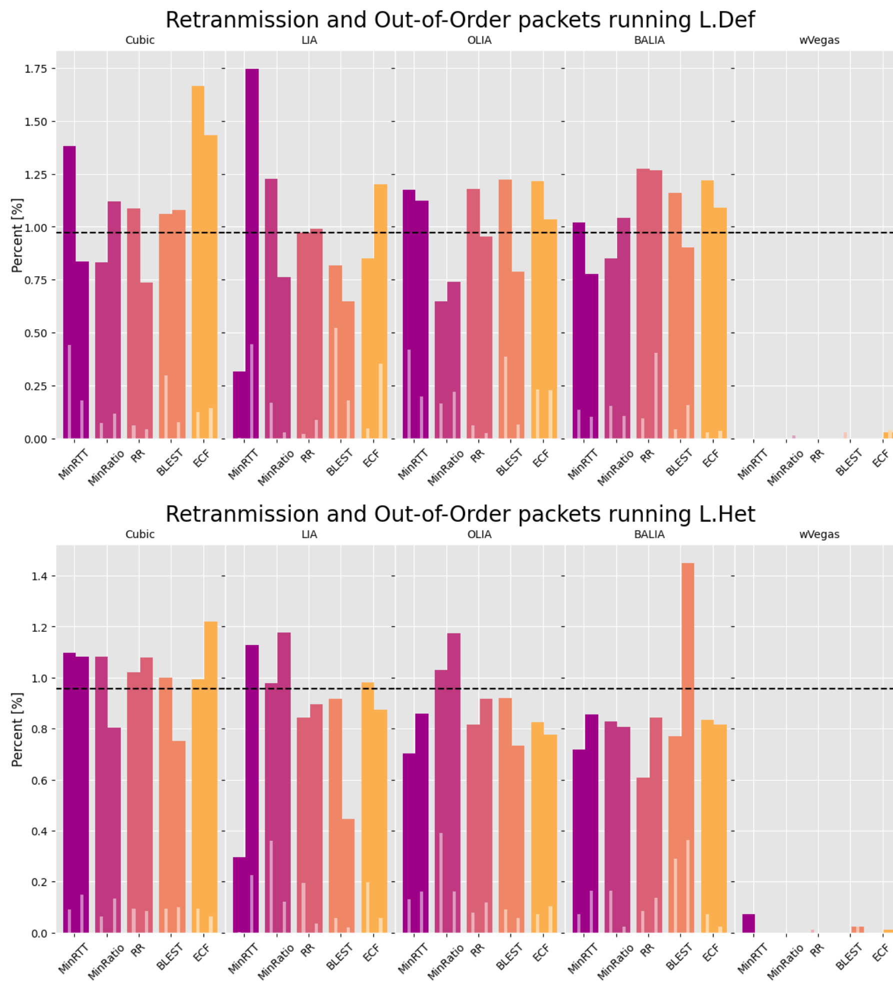

Figure 5.5: Mean percentage of detected Retransmissions (lighter, skinnier bars) and Out-of-Order packets (darker, thicker bars) by Wireshark for L.Def and L.Het with the dotted line showing the mean percentage of Out-of-Order packets for Cubic_MinRatio .

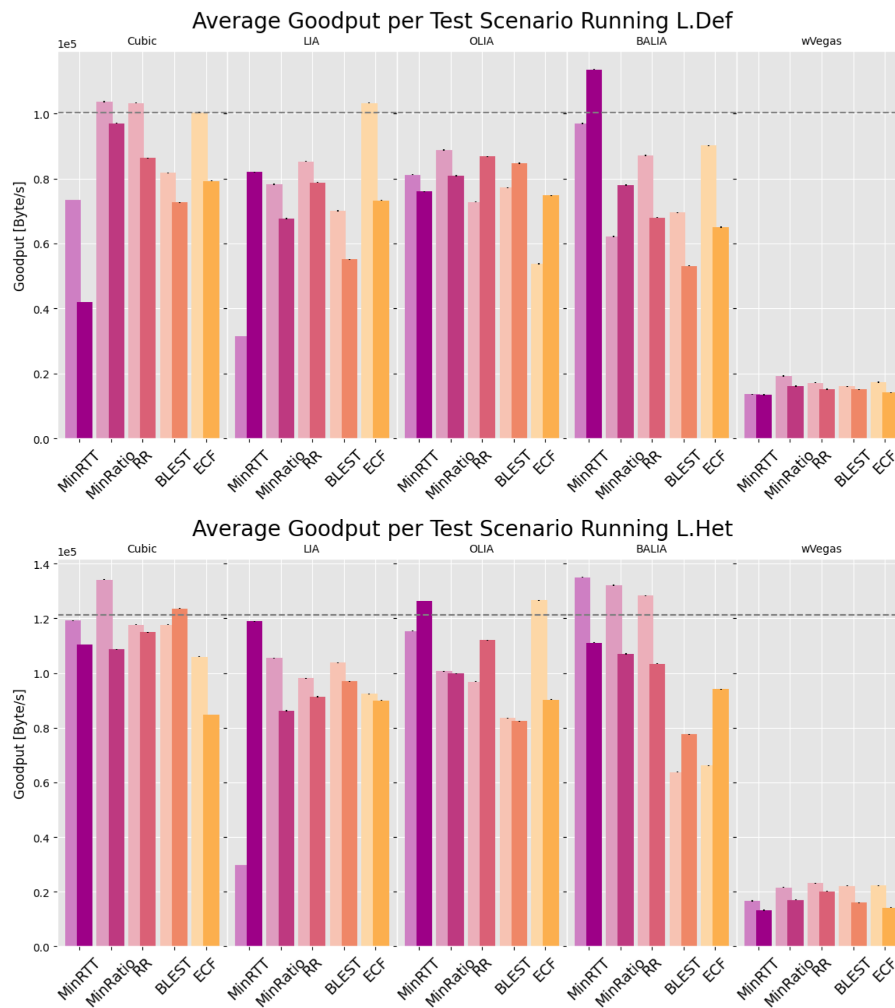

Figure 5.6: Average goodput per test scenarios L.Def and L.Het shows tests, mostly combinations utilising BALIA reaching the dotted baseline which marks the mean performance of the two subflows when running combination Cubic_MinRatio .

are among the better performers on the Live tests, if disregarding the combinations containing wVegas, while being evenly dispersed regarding standings in the Model tests (see Figure 5.13). This is explored in further detail in the next Section (5.3), while the differences between Live and Model performance are discussed further in Section 5.4.

5.3 Small Differences in Performance of Scheduling Algorithms

As seen in the mean latency aggregation Figure 5.3, there is no clear high or low performer regarding latency among the SAs as with the CCAs. Figure 5.4 confirms this initial observation. However, both figures show that MinRatio (baseline) and BLEST most frequently have the lowest latencies out of the five SAs, with BLEST average latency being the only one outperforming the baseline, if only by a few milliseconds. The margins between their performance could be considered insignificant, especially when considering their large standard deviations. The small differences are quite expected given that it is mainly the CCAs role to send as many packets as possible, while SAs rather fine-tune the results by attempting to utilise the subflows resources effectively and achieve in-order arrival.

As the schedulers main role is to decide on which subflow to send the data on it is interesting to split this data per subflow when reviewing it. This can be seen in Figure 5.5 which show retransmissions and Out-of-Order (OFO) packets per algorithm and subflow. The top graph in the figure shows that MinRTT, especially in combination with Cubic or LIA, schedules most retransmissions and measured Out-of-Order packets on one subflow. MinRTTs uneven utlisation of its subflows is also true for goodput, which is seen in Figure 5.7. Otherwise, the Figures show that the other algorithms keep an even divide between utilising both subflows in most of the CCA combinations. At a glance BLEST seems to be best at balancing, but more data would be necessary to conclusively say that one SA is better than the others at this task. To see how the algorithms fared while running on the same vs separate beams, refer back to Figure 5.6.

5.4 Varying Results Between Model and Live Testbeds

Most results discussed until this point have been from the tests running on the Live testbed, this is as they differ significantly from the tests run on the Model testbed as seen in all metrics. The decision to prioritise the Live testbed was done based on the unexpected results the Model testbed produced, both on the reference TCP tests, see Section 5.1, and MPTCP test which caused uncertainty regarding the result's validity.

Comparing latency results on the two testbeds in Figure 5.8 we see that the results are in the same size range which is reassuring. The results for wVegas are close

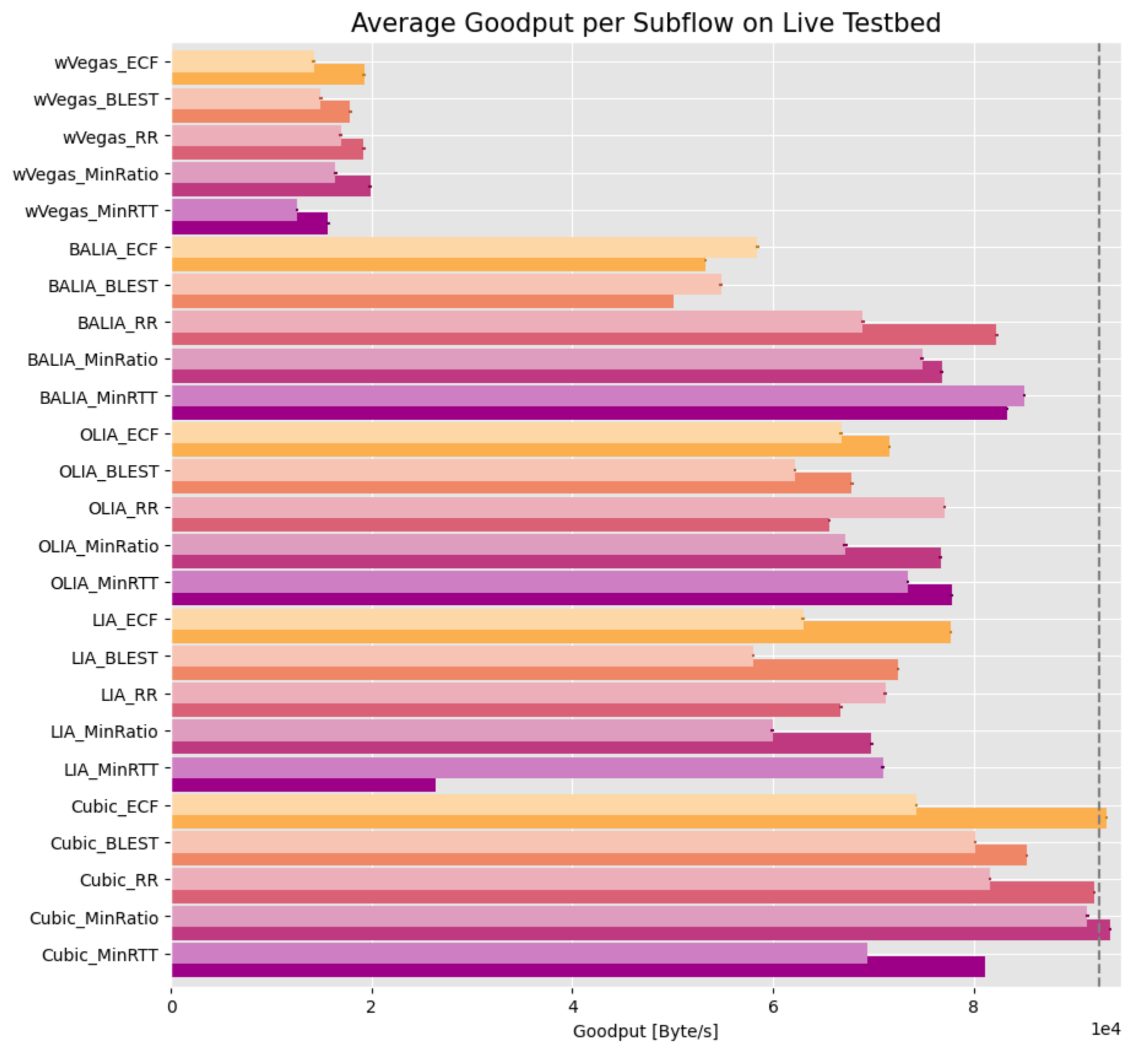

Figure 5.7: Aggregation of average goodput [Byte/s] divide per subflow for all test types per algorithm combination (large is better) when running iPerf3 on model testbeds.

on both testbeds, with an approximate difference of 0.3 s for each. However, the highest mean latency is approximately 2.3 and 4.8 s on the Model and Live testbed respectively, with the Model testbed outperforming the live tests significantly in this regard. It is clear that most algorithms have a much higher latency on the Live testbed compared to itself on the Model testbed. Overall, the variance in latency is a lot higher in the Live testbed than in the Model, with the difference between the highest and lowest latency being approximately 0.8 s on the Model testbed, and 2.6 s on the Live.

An initial hypothesis may be that this was caused by the required encapsulation in a UDP tunnel on the Live testbed, as mentioned in Section 4.2.3. The absence of WireGuard encapsulation could then be the reason that the Model testbed showed better performance. However, this hypothesis is not supported. As seen in the reference TCP data in Table 5.1, the Model testbed without WireGuard has equally bad performance as the Live testbed with WireGuard.

Instead, the differences between the testbeds may be explained by the difference in emulated and real link behaviour, as discussed in Section 5.1. There is a possibility that the set-up of destructive network behaviour running on the Network Emulator is oddly formatted, and consequently our modelled network behaviour do not accurately reflect reality. This would also explain the difference in throughput behaviour between the testbeds. The large differences are visualised in the Figure 5.10. The graph plots the throughput over time using data collected by SAR for the tests M.Def and L.Def.

A large limiter to throughput is the internal Congestion Window (CWND) set by the CCA, which is mainly decided by packet loss for Cubic, LIA, OLIA, and BALIA which are loss-based Congestion Controls, and delay in the case of wVegas. The emulated packet loss of 1% might be too high to accurately emulate the specific SATCOM network used in our Live testbed, but this is hard to confirm using the available data. However, we can view the send window showing how much available space the receiver has, which is often used by algorithms to not flood the receiver. This is plotted in Figure 5.9 in which we see that the send window for the model testbed at time dips quite low.

Also, as mentioned in Section 5.1, the actual implementation of the packet-loss may be unrealistically consistent. One of our hypotheses, suggested by our industry supervisor, is that that the Jalapeños buffer space was too small, which would result in a higher packet-loss than designed. The too high and consistent packet-loss could force a lower CWND and throughput, as seen in the graph for M.Def in Figure 5.10. However, by observing that the retransmissions are fewer on the Model testbed than on the Live as shown in Figure 5.11, one might argue that the emulated packet-loss was not too high, but rather to small, since higher packet-loss would correspond to more retransmissions. It is then worth noting that the data for that Figure is gathered using Wireshark at the server. Packets that are lost in transit to the server and retransmitted will not be recorded as a retransmission by Wireshark since the packet has not been seen before by the server. This means that in Figure 5.11, the retransmissions can not be directly liked with packet-loss.

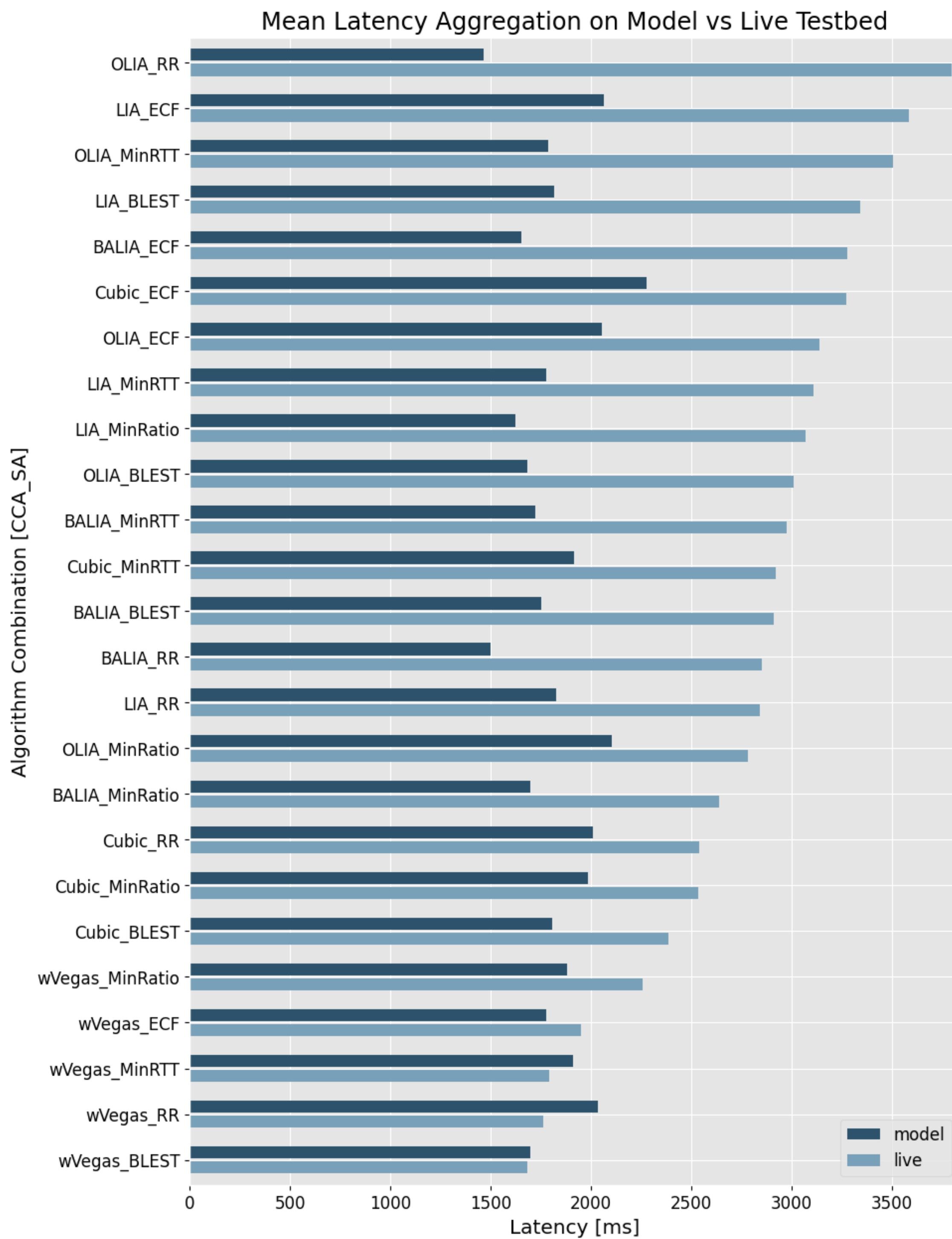

Figure 5.8: Mean latencies from aggregation of Sockperf under-load results on the Model vs Live testbed. Sorted by the Live testbed latencies in descending order.

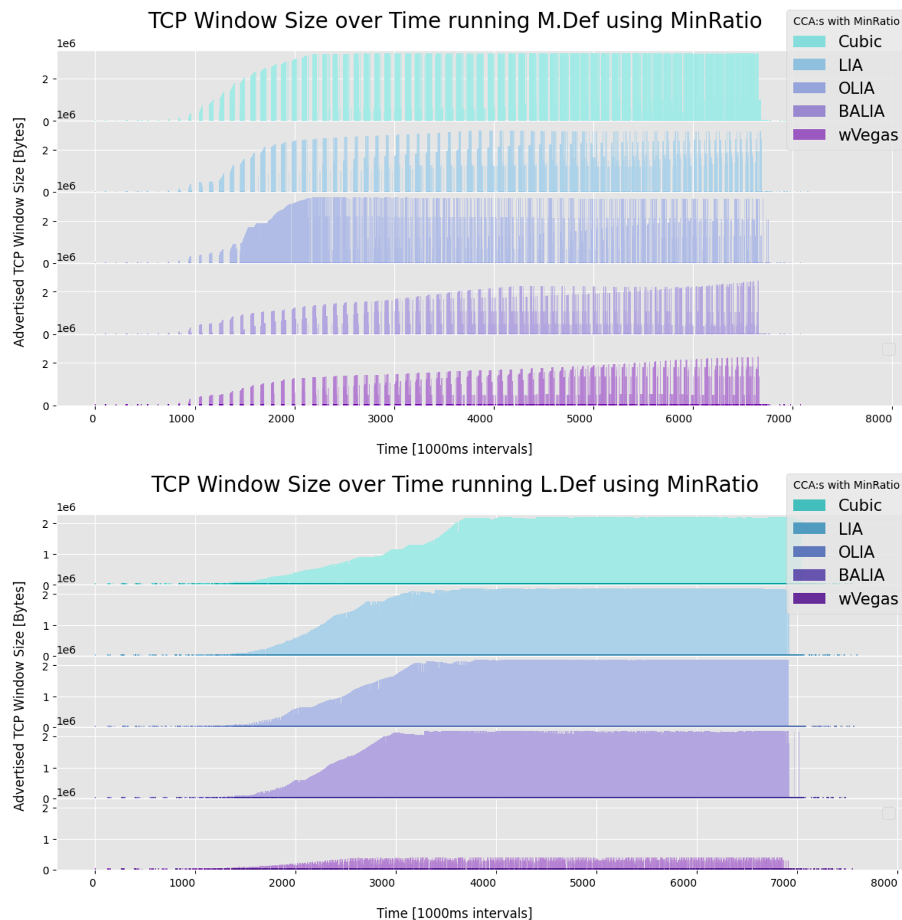

Figure 5.9: Plot of Send Window size per time unit for 5 test runs utilising the baseline scheduler MinRatio shows how well the CCA maintains a high send window (large is better). The results shown are as sent from iPerf3. Different colours are representative of different IP sources, with Live testbed having three due to the WireGuard configuration.

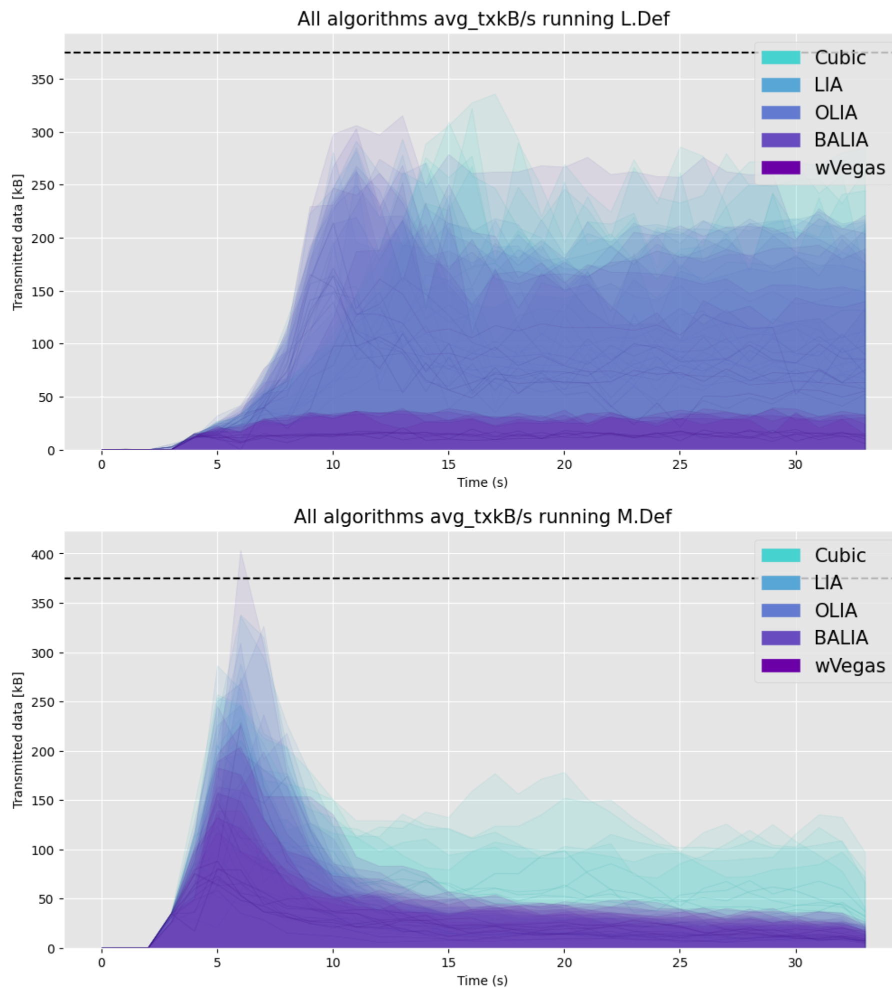

Figure 5.10: Plot of average transmitted Bytes/s for L.Def and M.Def shows that Live testbed (top) has a more even distribution of throughput throughout the test, while Model (bottom) testbed shows a sharp decline after the initial Slow Start. Data measured through SAR data on the network interface. Dotted line shows the theoretical bandwidth capacity.

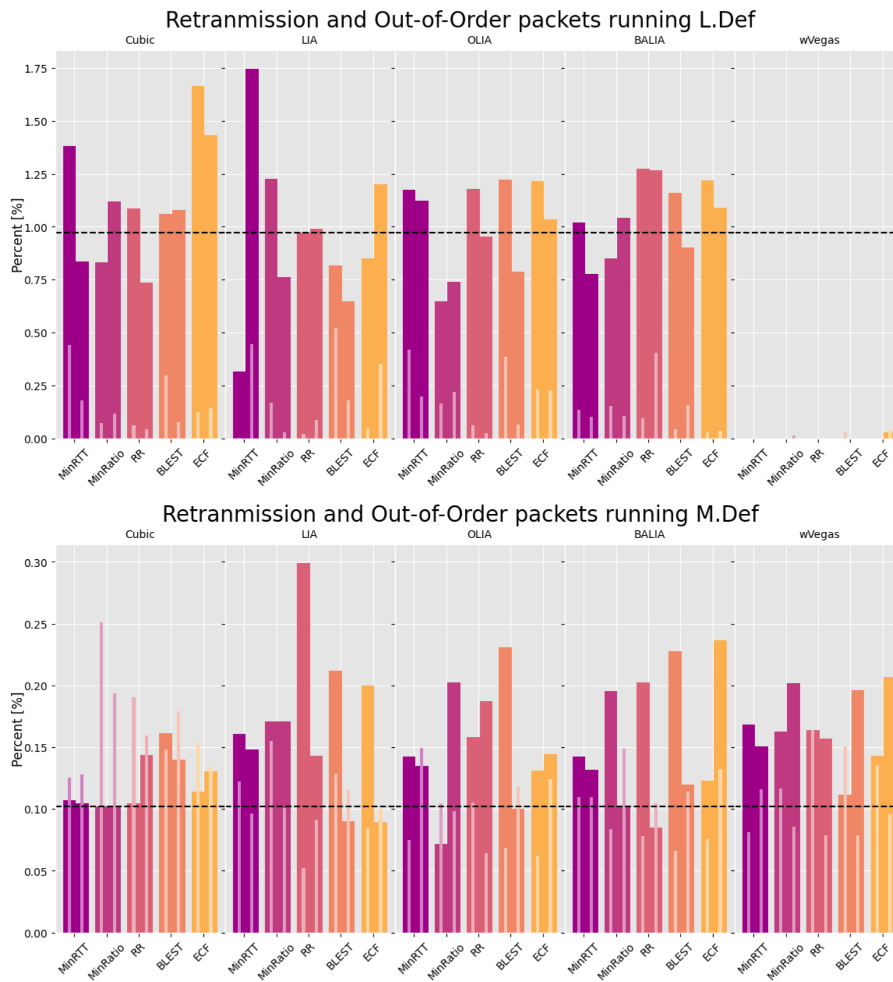

Figure 5.11: Mean percentage of detected Retransmissions and Out-of-Order packets by Wireshark for L.Def and M.Def shows that Live test bed has a higher level of packets marked as Out-of-Order (darker, thicker bars) than Model testbed while retransmissions (lighter, skinnier bars) remain comparable.

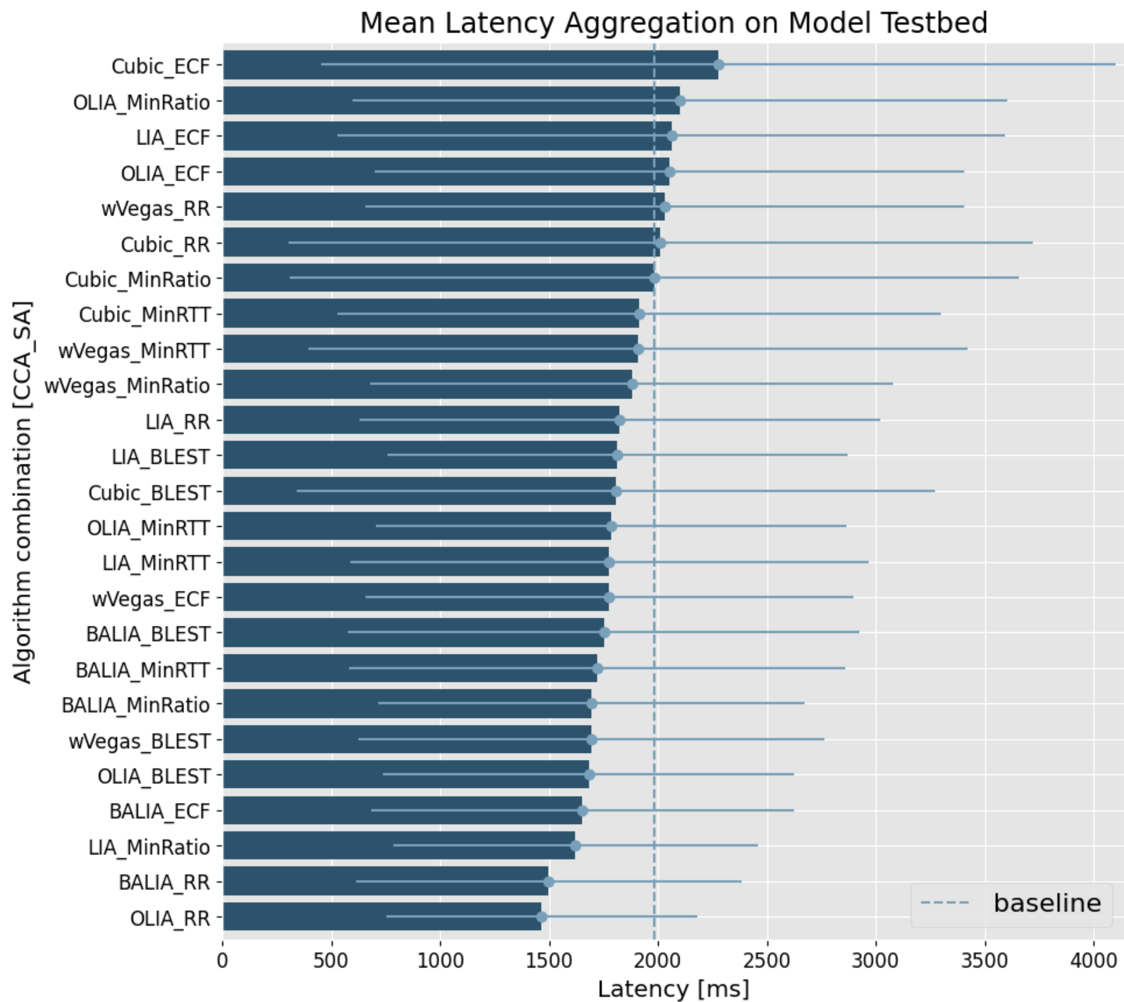

Figure 5.12: Mean latencies, with standard deviations, from aggregation of 20 Sockperf under-load tests per testcase on the Model Testbed. The baseline marks the mean latency of Cubic_MinRatio .

The difference in results between the Model and Live testbed brings into question the validity of the measurements. This is explored further in the next Section 5.5 concerning the validity of latency measurements, as well as in a separate Section 5.6 regarding validity of throughput measurements.

5.5 Validity of Latency Measurements

The observed performance was worse than expected, which put the result's validity into question. The aggregated one-way mean latencies can be seen in Figure 5.12 and 5.3 for the Model and Live testbeds respectively. The figures show a mean lowest latency of approximately 1.5 s on both the Model and Live testbed. This is surprisingly high since the estimated one-way latency of a satellite link was 250 ms. The mean highest latency seen in the figures are approximately 2.3 on the on the Model and 4.8 on the Live testbed, which is 9.2 and 19.24 times higher

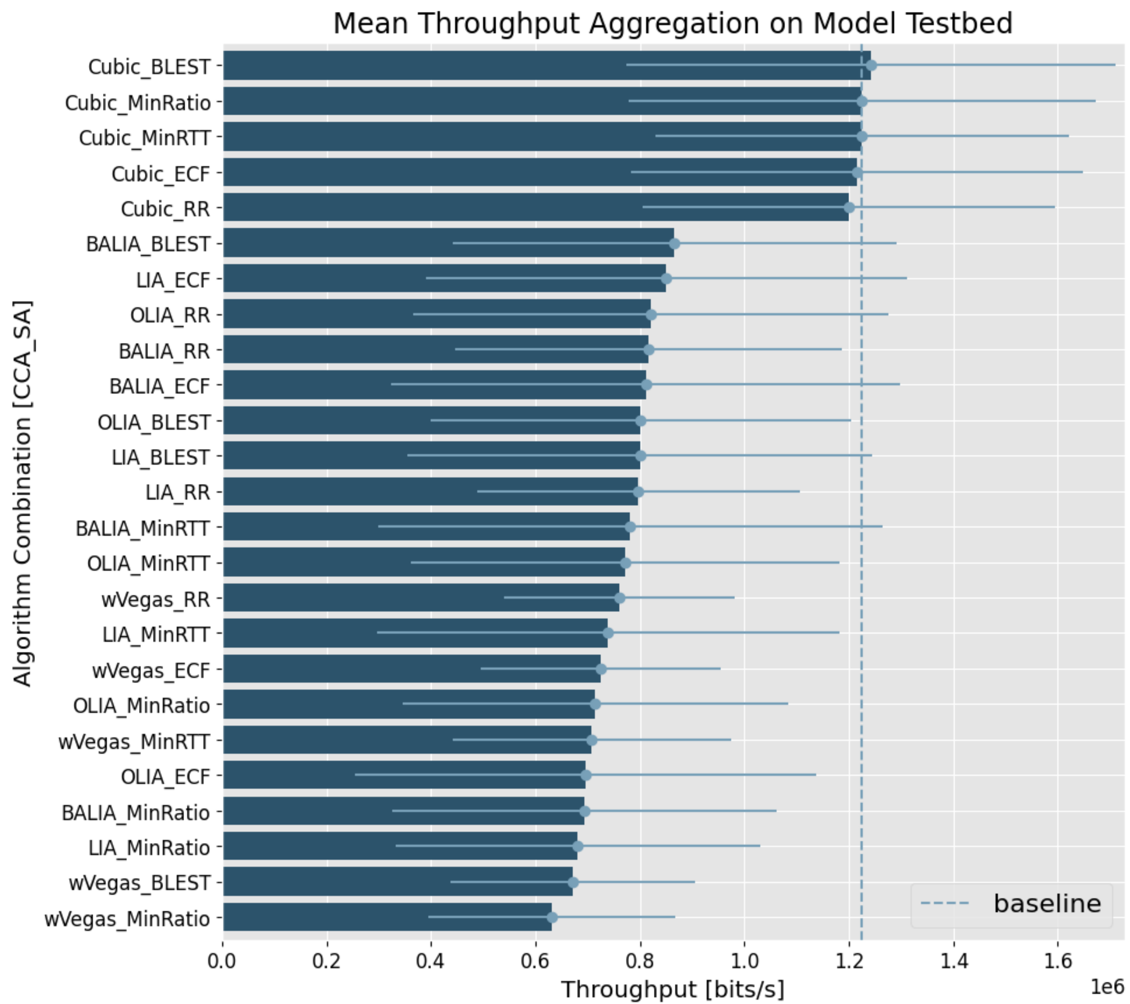

Figure 5.13: Mean throughput, with standard deviation, from aggregation of iPerf3 results on the Model Testbed. The baseline marks the mean throughput of Cubic_MinRTT.

than the estimation, respectively. However, as seen in Table 5.1 the magnitude of the mean latency using TCP on the Model testbed and on the Live testbed with WireGuard align with that observed when using MPTCP. This strongly supports that the MPTCP results are valid. The cause of the high latency values for both TCP and MPTCP are discussed in Section 5.1.

Doubt regarding the result's validity may also arise from the major differences in algorithms performance relative to one another on the Model vs the Live testbed. This is seen when comparing the latency results of the Model testbed in Figure 5.12 to the Live in Figure 5.3. As discussed in Section 5.4, this could indicate that there are a behaviours on the Live testbed which the Model testbed has not emulated correctly, either due to the tools limitations or something lacking in our configuration. The issue have already been expanded on in that section and thus will not be repeated. However, it is worth mentioning that due to these differences, the Live testbeds should be considered a better reflection of the true performance.

5.6 Validity of Throughput and Goodput measurements

By comparing the aggregated mean throughputs with standard deviations as can be seen in Figure 5.13 and 5.2 for the Model and Live testbed respectively for which no mathematical outliers were found with the expected bandwidth of the link is a good starting point for validity checking of the tests. The bandwidth of the links lies around 3 Mbps for uplink and 10 Mbps for downlink, with the weaker of the links being the limiting factor in our tests. When put into such a context the mean throughput of just over 1.5 Mbps on the Live test bed seems quite low. However, it is a quarter of what would be perfect bandwidth aggregation, which would require no other data sent on the links which is not impossibly low. This low throughput is, however repeated on the Model testbed with even worse results. Although that could be an issue with modelling.

The low performance of the Model testbed in regard to throughput is perhaps most clear when looking at Figure 5.10 where we see a sharp decline after initially keeping a high bandwidth. That behaviour can be expected due to the explosive Slow Start of CCAs which aim to mellow down in order to avoid packet losses. However, given that most tests on the Model testbed experienced little to no competing data on most parts of the link, it is quite strange, as discussed further in Section 5.4. The Live testbeds throughput does however look more promising when plotting the tests in the format of Figure 5.10. On the graph we see the bandwidth plotted as a dotted line at 375 kBps, which is simply a conversion of the 3 Mbps uplink. Although the measured average throughput does not quite reach this value, but achieving over 300 kBps at points, with some test iterations for Cubic and BALIA somewhat consistently remaining at 250 kBps. Given that the tests are live, with us having no control over what other data is sent over the beams, and other processing done at the satellite this is quite promising.

6
Conclusion

The goal of our thesis was to conduct a comparative study of Congestion Control (CCA) and Scheduling (SA) Algorithms specifically within meshed, satellite Multi-WANs. Thus exploring possible improvements and trade-offs done when choosing algorithms for one's network. Within our scope we chose to limit ourselves to explore geostationary (GEO) satellites and MPTCP in order to limit the amount of work and testing required. Although the end result did not show some specific algorithms greatly outperforming others that was not necessarily expected. Especially as the algorithms we chose which could be found within the v0.96 Linux Kernel as mentioned in Section 3.1, hence them being among the older algorithms suggested in the sense that they were realised pending the release of the first in-tree release of MPTCP. Of course, except Cubic and MinRatio which have been the default choice for MPTCP recently [19], [20].

6.1 Notable Results

The results presented in the Chapter 5 indicates that the current default choice Cubic_MinRatio outperforms the other tested algorithm combinations in the specific case of GEO satellite communication. However, it is worth noting that although it consistently showed a slight edge over the other configurations it is quite possible that even small changes in the network environment such as a move to some other location connecting to some other beam or even satellite could change those outcomes. If we had more time, and the Model results were more similar to those of the Live testbed this could have been explored through adjustment of parameters.

It is also worth mentioning wVegas highly outperforming other algorithms in terms of reliably delivering packets with low latency, and in-order. This showcases its usefulness in more specific use cases, such as critical information sharing, and jitterless transmission for video streaming, which were mentioned as possible more specific uses of satellite communication in Section 2.1.

Although not directly stated as a hypothesis in the Experiment Design Chapter 3 we had expected Cubic to be among the best CCAs due to its uncoupled nature. As shown in Chapter 5, this turned out to be the case as Cubic strongly outperformed all other CCAs regarding throughput and goodput. We also expected Round Robin to be the worst SA by far due to how it is mentioned in [17] which claim that it has no value being used in real applications due to its poor performance. However,

as seen in Section 5.3 that was not the case in our tests where it often appeared among the top half of performers. We believe that Round Robin gets an edge over other algorithms due to having to make no calculations in its scheduling decision, especially as our testcases only utilised two subflows.

6.1.1 Concerns Regarding SATCOM Network Emulation

As frequently mentioned in Chapter 5, the results often differ significantly between the Model and Live testbed, This could either be indicative of a poorly emulated network environment, or simply being representative of some other possible SATCOM network. Given that it was not within the scope of the thesis to perform such measurements we had to assume that the Model tests are not reliable and instead look to the Live tests for our conclusions.

However, this raises the question of to what lengths results run on modelled SATCOMS can be trusted. The network configurations used on our Model testbed was selected due to similar values being used in previous SATCOM research. Our thesis has showed the importance of tests on live networks, even though it comes with its limitations. Further investigating the reasons behind the different behaviour of our Model testbed compared to a live network may be of interest for actors such as Satcube who wish to continue modelling SATCOM environments. Some hypotheses have been discussed in this thesis, such as unrealistic formatting of the packet-loss or the lacking buffer-space in the Jalapeño. These may act as starting points for further investigation.

6.1.2 The Limitations of Live Tests

The decision to perform live tests was quite limiting in terms of kernel version used, as well as making up a large portion of the implementation time. It also limited us in terms of hardware and amount of subflows which we could possibly connect. Although it turned out to have been a sound decision, especially as it showed how different our results were between the Model and Live tests in Section 5.4, indicating that our Network Emulation as described in Section 4.1.3 was too far off.

Further choosing to perform live tests tightened our time constrictions for the analysis part of the project by a lot as it was the leading cause of the limitation of Linux kernel version we had to use as mentioned in Chapter 3 and Section 4.2. We had in researching our code assumed that most of the underlying logic would be the same between kernels, however large refactoring had been made in between the versions in which data-structures for TCP and MPTCP had been changed. A part of which was changing the scheduling of packets to a queue instead of bitmasking subflows to send a packet on. This ultimately lead to the exclusion of testing Redundant SA, as segments could only be scheduled to a singular subflow and not multiple without complete rework of the packet scheduler in kernel version v.5.15.60 [18], [20].

As mentioned a large contribution to the changes to our plans made was due to lack of time. With the misjudgement to how long it would take to re-implement the algorithms in the Linux kernel being the largest contributor taking over two months

instead of some weeks as estimated had the implementations been more similar. Due to this the testing was not as incremental as we would have liked. Further we had little to no time to adjust the model settings after seeing the live results. Otherwise, they would have probably been adjusted to be truer to the measured live tests, which would have allowed us to make reasonable guesses on how more largely heterogenous links might appear. This would have allowed for more truthful heterogenous testing than what we achieved, given that our Model and Live tests are not fairly comparable. Further given more time, more thorough investigation into our results regarding the longer than expected latencies, and lower throughputs could have been made. In order to make such analysis it would have been healthy to have more comparisons to regular TCP traffic as well in combination to the MPTCP tests.

6.2 Possible Improvements

As to possible future work to build further on this project there are multiple improvements which could be made. This includes additional testcases and metrics that had to be cut due to lack of time. These, as well the impact of including slow start in our measurements are discussed in this Section.

Measure Flow Completion Time: Further we wished to evaluate Flow Completion Time (FCT) as it felt like an interesting metric for some of the use cases of specifically satellite terminals with limited batteries, but also as it ties together the different metrics into a tangible format. For example, it could be easier to understand the impact of algorithms if we could state that transmitting a 1 GB video would take some amount of seconds depending on the algorithms. Further it could help show whether some trade-offs such as lower throughput for the sake of decreased latency is worthwhile, such as the case for using the CCA wVegas as seen in Section 5.2. It was cut due to difficulties in data collection as although our chosen testing tools sockperf and iPerf could send batched data, the data itself was not split into different data streams. Which meant that we would have had to create some new version of a testing tool in order to get useful measurements of this.

Measure CWND: Another metric which is only inferred in our test results is the Congestion Control Window, or CWND for short. This metric is internal to the CCA and is not advertised on the MPTCP connection like the send window plotted which only shows how much data the receiver can handle as plotted in Figure 5.9. However, as it is the main steering factor of CCAs, it could have been interesting to plot. Although it can be inferred by looking at throughput, or more correctly the amount of bytes in flight, as that is what the CWND is meant to limit as described in other words in Section 2.2.2. Due to it not being an advertised metric, it would have required logging the value from the kernel code or finding a tool which could have done so for us. Given the timeline of our project had already been challenged by implementation we did not attempt collecting the data.

Exclude Slow Start: Our choice to include the Slow Start stage present in all CCAs would lead to throughput and latency which at a glance seems to perform worse than reported in other papers, which generally exclude it from their results. This is as all congestion controls by design start with small congestion windows and rapidly increase it which possibly leads to losses. The decision to disregard that period in many papers is likely done due to an assumption that it is not part of the normal use of the algorithms, assuming long, unchanging connections. However, much browsing and download lead to quite short bursts of data in loading in a webpage and likely connecting to another server to load in other webpages. This of course is not true in the case of streaming data and large downloads such as the case of longer format video data.

Test with Different Data Structures: Something which this thesis did not explore, is how different appearances of data, with the general packet size, and burst size of transmitted data affects the performance of the algorithms. By including different appearances of the transmitted data one would have more support to make claims what use cases they are good for.

6.2.1 Additional testcases:

Initially we had planned to perform more testcases than we found feasible after getting further into the thesis. Most notably cutting dynamic tests, as seen in this list of previously planned tests that were not run:

- **Dynamic removal:** The default testcase, except one subflow is removed during transmission.
- **Dynamic join:** The default testcase but with only one subflow at the start. A second subflow is made available during transmission.
- **Simultaneous sending on heterogenous paths:** Both PBs sending on two shared heterogenous paths, using two subflows per PB. One PB sending a constant stream of data, using the baseline SA and CCA The other PB performs test transmissions. Both passively forwards the other PB's transmissions.
- **Heterogenous paths on separate satellites:** Similar to L.Het, but the two paths connect to separate satellites, instead of separate beams but from the same satellite.

The dynamic join and removal would have helped us assess the algorithms' adaptability to dynamic network environments, such as when beams are obstructed more than just briefly. Simultaneous sending on heterogenous paths would have provided a more well-rounded picture of the algorithms' performance on shared paths. These testcases were however cut early, mainly due the time constraint which caused us to reduce the amount of testcases. The decision to cut the dynamic testcases specifically was made since we had no strategy in place for how they would be implemented. Exploring the behaviour on homogenous, heterogenous, and shared links had more importance for this thesis aim, and was thus more worth our time. Heterogenous

paths on separate satellites were cut due to limitations of our geography, only having one GEO satellite which we could reach from the office.

Additionally, we suspect that the use of maximum two subflows contributed to the inconclusive results seen when comparing different Scheduling Algorithms (SAs). When the SAs only have two subflows to choose between, their behaviour end up rather similar. When one subflow is filled, there is only one remaining option, leading to many algorithms functioning in an adaptively weighted Round Robin fashion. Expanding the testcases to run on 3 or more subflows and more connected clients would probably give more interesting results regarding the SA performance.

Lastly, it would have been useful to run iPerf3 test using TCP in addition to the Sockperf under-load tests. As discussed in Section 5.1, Sockperf under-load tests are not technically designed to test throughput. iPerf3 tests would be more suitable for throughput measurements for TCP, but had to be cut due to lack of time.

6.2.2 Future Work

There are still areas regarding SATCOM and especially GEO SATCOM which can be explored further. For example this thesis looked into coupled CCA which prioritise fairness. As such an opportunity of expansion is looking into the possibly more performant uncoupled CCA. When exploring such algorithms it would be interesting to measure how other data sent on the subflow is affected, to show that it is not to other connections on the network. Work testing CCAs would benefit for inclusion of dynamic network conditions as mentioned as a possible improvement for our thesis as well in Section 6.2.1.

One could also explore more packet SA, in which testing with a varying number of paths, and link characteristics would be beneficial. Such tests would be benefitted by having three or more possible subflows as described as a possible improvement in Section 6.2.1. Further the FCT metric would be an interesting addition to both tests of more CCA and SA. However, it would be our recommendation to choose to test only CCA or only SA at a time to have less data to compare. Especially as we in this thesis saw that synergistic or detrimental relationships between the two was rare in this thesis, we feel little would be lost if the testing had been split.

6.3 Final Words

To reiterate this thesis aimed to investigate the performance of various Congestion Control and Scheduling Algorithms in the context of MPTCP over GEO satellite links. While the results did not reveal a single, overall superior algorithm, they highlighted interesting observations. With BALIA_MinRTT exhibiting promising performance in Live tests, even surpassing the default Cubic_MinRatio in terms of throughput. Which given the nature of coupled CCAs being more suppressing to themselves in order to promote fairness on the links to regular TCP traffic, at their own decrement. Meanwhile, wVegas demonstrated reliability of low latency delivery, without needing retransmissions, surpassing the performance of Cubic in

that aspect by a large. With some surprising results such as Round Robin being more performant in the simple network environment we tested in than given the expectation to by the developers.

With the results we would also state that more comprehensive testing is needed. Importantly through including testing of dynamic links to assess the algorithms' adaptability to changing network conditions present in real-world usage. Further we felt that including metrics such as Flow Completion Time analysis would help us better understand the overall impact of algorithms for the users of the network. This alongside a more true to life network emulation, or live tests performed in more environments would be helpful to provide more robust insight to the use of MPTCP in GEO satellite networks.

Git repository with a patch with our changes can be found in: `https://github.com/katri-lantto/mptcp-thesis`.

Bibliography

[1] *Satellite Glossary | ABS Global Ltd.* [Online]. Available: https://absatellite.com/support/satellite-glossary/.

[2] L. J. Ippolito, *Satellite Communications Systems Engineering*. Hoboken, NJ: Wiley, Sep. 2008, ISBN: 9780470725276. DOI: 10.1002/9780470754443.

[3] M. A. Yousuf, M. M. Islam, M. S. Hosen, and M. L. Ali, "Round-Trip Time and Available Bandwidth Estimation Based Congestion Window Reduction Algorithm of MPTCP in Lossy Satellite Networks," in *Journal of Physics: Conference Series*, vol. 1624, 2020. DOI: 10.1088/1742-6596/1624/4/042024.

[4] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC Editor, Tech. Rep., Mar. 2020. DOI: 10.17487/rfc8684.

[5] C. Raiciu, M. Handley, and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols," Tech. Rep., Oct. 2011. DOI: 10.17487/rfc6356.

[6] Y. Xing, K. Xue, Y. Zhang, *et al.*, "A Low-Latency MPTCP Scheduler for Live Video Streaming in Mobile Networks," *IEEE Transactions on Wireless Communications*, vol. 20, no. 11, pp. 7230–7242, Nov. 2021, ISSN: 1536-1276. DOI: 10.1109/TWC.2021.3081498.

[7] R. Chen, W. N. Wang, J. L. Zhu, and B. Wang, "A round-trip-time based concurrent transmission scheduling for MPTCP," in *2014 6th International Conference on Wireless Communications and Signal Processing, WCSP 2014*, 2014. DOI: 10.1109/WCSP.2014.6992036.

[8] F. Yang, Q. Wang, and P. D. Amer, "Out-of-Order Transmission for In-Order Arrival Scheduling for Multipath TCP," in *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, IEEE, May 2014, pp. 749–752, ISBN: 978-1-4799-2653-4. DOI: 10.1109/WAINA.2014.122.

[9] A. Axelzon and L. Norman, "Enhancing Satellite Communication Performance with MultiWAN using MPTCP: Implementation and Analysis Improving Bandwidth Efficiency and Reliability in Satellite Communication," Chalmers University of Technology, Gothenburg, Tech. Rep., 2023. [Online]. Available: https://odr.chalmers.se/server/api/core/bitstreams/17319ec6-71fb-42d2-a102-1f98c954c839/content.

[10] Satcube, *Satcube Ku*.

[11] European Space Agency, *Satellite frequency bands*. [Online]. Available: https://www.esa.int/Applications/Connectivity_and_Secure_Communications/Satellite_frequency_bands.

[12] *A straightforward introduction to satellite communications*. [Online]. Available: https://www.inmarsat.com/en/insights/corporate/2023/a-straightforward-introduction-to-satellite-communications.html.

[13] K. Sergieieva, *Types Of Satellites: Different Orbits & Real-World Uses*, Mar. 2023. [Online]. Available: https://eos.com/blog/types-of-satellites/.

[14] *MPTCP | Multipath TCP for Linux*. [Online]. Available: https://www.mptcp.dev/.

[15] W. Eddy and Ed., "Transmission Control Protocol (TCP)," Tech. Rep., Aug. 2022. DOI: 10.17487/RFC9293.

[16] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," Tech. Rep., Sep. 2001. DOI: 10.17487/rfc3168.

[17] Multipath TCP community, *MultiPath TCP - Linux Kernel implementation : Users - Configure MPTCP browse*, 2021. [Online]. Available: https://multipath-tcp.org/pmwiki.php/Users/ConfigureMPTCP.

[18] *multipath-tcp/mptcp*, Feb. 2023. [Online]. Available: https://github.com/multipath-tcp/mptcp/commits/v0.96/.

[19] *multipath-tcp/mptcp_net-next*, Nov. 2024. [Online]. Available: https://github.com/multipath-tcp/mptcp_net-next.

[20] *torvalds/linux v5.15.60*, Aug. 2022.

[21] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure, "Experimental evaluation of multipath TCP schedulers," in *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, New York, NY, USA: ACM, Aug. 2014, pp. 27–32, ISBN: 9781450329910. DOI: 10.1145/2630088.2631977.

[22] Y.-s. Lim, E. M. Nahum, D. Towsley, and R. J. Gibbens, "ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, New York, NY, USA: ACM, Nov. 2017, pp. 147–159, ISBN: 9781450354226. DOI: 10.1145/3143361.3143376.

[23] B. Y. L. Kimura, D. C. S. F. Lima, and A. A. F. Loureiro, "Alternative Scheduling Decisions for Multipath TCP," *IEEE Communications Letters*, vol. 21, no. 11, pp. 2412–2415, Nov. 2017, ISSN: 1089-7798. DOI: 10.1109/LCOMM.2017.2740918.

[24] F. Yang, P. Amer, and N. Ekiz, "A Scheduler for Multipath TCP," in *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*, IEEE, Jul. 2013, pp. 1–7, ISBN: 978-1-4673-5775-3. DOI: 10.1109/ICCCN.2013.6614091.

[25] S. Ferlin, O. Alay, O. Mehani, and R. Boreli, "BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks," in *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, IEEE, May 2016, pp. 431–439, ISBN: 978-3-9018-8283-8. DOI: 10.1109/IFIPNetworking.2016.7497206.

[26] N. Kuhn, E. Lochin, A. Mifdaoui, G. Sarwar, O. Mehani, and R. Boreli, "DAPS: Intelligent delay-aware packet scheduling for multipath transport," in *2014 IEEE International Conference on Communications (ICC)*, IEEE, Jun. 2014, pp. 1222–1227, ISBN: 978-1-4799-2003-7. DOI: 10.1109/ICC.2014.6883488.

[27] K. W. Choi, Y. S. Cho, Aneta, J. W. Lee, S. M. Cho, and J. Choi, "Optimal load balancing scheduler for MPTCP-based bandwidth aggregation in heterogeneous wireless environments," *Computer Communications*, vol. 112, pp. 116–130, Nov. 2017, ISSN: 01403664. DOI: 10.1016/j.comcom.2017.08.018.

[28] P. Hurtig, K.-J. Grinnemo, A. Brunstrom, S. Ferlin, O. Alay, and N. Kuhn, "Low-Latency Scheduling in MPTCP," *IEEE/ACM Transactions on Networking*, vol. 27, no. 1, pp. 302–315, Feb. 2019, ISSN: 1063-6692. DOI: 10.1109/TNET.2018.2884791.

[29] C. Raiciu, C. Paasch, S. Barre, *et al.*, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, USENIX Association, Apr. 2012.

[30] D. Hayes and et.al, "Report on prototype development and evaluation of end-system, applica- tion layer- and API mechanisms," Simula Res. Lab, Oslo, Tech. Rep., Sep. 2015, pp. 17–19.

[31] B. Y. L. Kimura and A. A. F. Loureiro, "MPTCP Linux Kernel Congestion Controls," Dec. 2018.

[32] F. Jowkarishasaltaneh and J. But, "An Analysis of MPTCP Congestion Control," *Telecom*, vol. 3, no. 4, pp. 581–609, Oct. 2022, ISSN: 2673-4001. DOI: 10.3390/telecom3040033.

[33] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks," Tech. Rep., Feb. 2018. DOI: 10.17487/RFC8312.

[34] Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec, "Opportunistic Linked-Increases Congestion Control Algorithm for MPTCP," *nternet Engineering Task Force, draft-khalili-mptcp-congestion-control-05*, Jul. 2014.

[35] Anwar Walid, Qiuyu Peng, Jaehyun Hwang, and S. Low, "Balanced Linked Adaptation Congestion Control Algorithm for MPTCP," *Internet Engineering Task Force, Internet-Draft draft-walid-mptcp-congestion-control-04*, Jan. 2016.

[36] Yu Cao, Mingwei Xu, and Xiaoming Fu, "Delay-based congestion control for multipath TCP," in *2012 20th IEEE International Conference on Network Protocols (ICNP)*, IEEE, Oct. 2012, pp. 1–10, ISBN: 978-1-4673-2447-2. DOI: 10.1109/ICNP.2012.6459978.

[37] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," Tech. Rep., Sep. 2009. DOI: 10.17487/rfc5681.

[38] Hendrik Cech, "Analyzing and Realizing Multipath TCP Schedulers in Linux," Technical University of Munich, Tech. Rep., 2020.

[39] *MultiPath TCP - Linux Kernel implementation : Main - Home Page browse.* [Online]. Available: https://www.multipath-tcp.org/.

[40] *MPTCP | Multipath TCP for Linux Introduction.* [Online]. Available: https://www.mptcp.dev/.

[41] *SockPerf*, Sep. 2022.

[42] *NetworkManager.* [Online]. Available: https://networkmanager.dev/.

[43] 8devices, *Jalapeno*, 2024. [Online]. Available: https://www.8devices.com/products/jalapeno.

[44] Bert Hubert, *tc(8) Linux manual page*, Dec. 2001.
[45] *Setup | Multipath TCP for Linux*. [Online]. Available: https://www.mptcp.dev/setup.html.
[46] *mptcpize - Man Page*. [Online]. Available: https://www.mankier.com/8/mptcpize.
[47] Jason A. Donenfeld., *WireGuard*, 2022.
[48] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu, *iPerf3*, 2024.
[49] Wireshark Foundation, *Wireshark*.
[50] Sebastien Godard, *sysstat - System performance tools for the Linux operating system*, 2024. [Online]. Available: https://github.com/sysstat/sysstat.
[51] *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. [Online]. Available: https://iperf.fr/.
[52] *Lua*, Jun. 2024. [Online]. Available: https://www.lua.org/.
[53] *TCPDUMP*, Aug. 2024. [Online]. Available: https://www.tcpdump.org/.