WhiteHaul-based Spectrum Aggregation in Wi-Fi Bands

Ka Wing Li

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2023

Abstract

In this paper, we examine an existing solution, WhiteHaul, about its effectiveness in aggregation multiple Wi-Fi channels in 5GHz spectrum. Our main focus is the performance of WhiteHaul in the presence of interference. In this report, we go through trials and errors to find areas of improvement and verify our hypothesis.

We begin by treating WhiteHaul as a black box and perform a baseline measurement of the aggregation throughput achieved by WhiteHaul with and without interference. We find that there is a significant decrease in total throughput when there is interference. Then we unbox WhiteHaul and hypothesise that the scheduler is causing the degradation in performance, but a subsequent experiment disproved our blief.

We move on to another component of WhiteHaul, congestion control, and conduct an experiment showing that congestion control impacts WhiteHaul's effectiveness. Then we get into the microscopic view of the congestion control and trace its state over transmission time. From the trace log, we discover an implementation problem for WhiteHaul inside the Linux kernel.

To sum up our findings, we derive an upper bound of throughput for any aggregation system and show that WhiteHaul performance is close to this upperbound. Based on the WhiteHaul source code, we proposed a new variant TCP Eindburgh and evaluated its performance against the original WhiteHaul. We also propose a list of potential alternative improvements.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ka Wing Li*)

Acknowledgements

I would like to thank my supervisor Mahesh Marina for his feedback, guidance and support. I also thank Dr. Mohamed Kassem and Dr. Tanya Shreedhar for their advice throughout.

Table of Contents

List of Figures

List of Tables

Acronyms

ACI Adjacent Channel Interference.

AIMD Additive-Increase/Multiplicative-Decrease.

BDP Bandwidth-Delay Product.

CBR Constant Bit Rate.

CCI Co-Channel Interference.

CSMA/CA Carrier-Sense Multiple Access with Collision Avoidance.

CWND Congestion Window Size.

HoL Head-of-Line.

MCS Modulation and Coding Scheme.

MPTCP Multipath Transmission Control Protocol.

RTT Round-Trip Time.

RWND Receiver Window Size.

SINR Signal-to-Interference-plus-Noise Ratio.

sRTT Smoothed Round-Trip Time.

SSTHRESH Slow Start Threshold.

TCP Transmission Control Protocol.

TDMA Time Division Multiple Access.

UDP User Datagram Protocol.

Chapter 1

Introduction

1.1 Motivations

Wireless technology has become an indispensable tool in modern communication systems, with an ubiquitous presence in almost every aspect of daily life because of its mobility. Multiple wireless interfaces have emerged as a standard feature in most electronic devices, and the ability to access these interfaces has revolutionised the way we communicate and access information. However, to fully utilize the benefits of multiple wireless interfaces, it is necessary to explore the potential of wireless aggregation. The seamless handover between LTE and Wi-Fi networks by Apple Siri, using Multi-Path TCP (MPTCP), is an encouraging example of using multiple interfaces for speed and reliability.

The increasing prevalence of dual-band and tri-band routers has prompted a significant research question: Can we fully utilize the bandwidth of multiple Wi-Fi channels by aggregating them? Instead of aggregating heterogeneous wireless technologies, we want to explore the potential benefits of wireless aggregation of the same technology, particularly in the context of utilising multiple Wi-Fi channels to achieve high throughput. Consider the following scenario; we have a dual-band router at home connecting to 1Gbps Ethernet, and our laptop has a built-in Wi-Fi card supporting Wi-Fi with a maximum throughput of 250 Mbps. If we have a USB Wi-Fi dongle at hand, can we plug it into my laptop and connect it to the second band of the router and achieve higher throughput?

The benefits of Wi-Fi aggregation are significant, including faster download and upload speeds, reduced latency, and improved reliability. However, the implementation of Wi-Fi aggregation also presents numerous challenges. Ensuring that aggregated data arrives at the destination in the correct order is one of the most significant challenges, requiring sophisticated designs and algorithms that can manage the flow of data across multiple paths. Additionally, the interference between different wireless channels is a dire concern, as it can impact performance if not managed properly.

WhiteHaul[23] is an efficient spectrum aggregation system for backhaul traffic over TV white space. It is promising to apply similar ideas to aggregate different Wi-Fi channels.

However, it has been reported that WhiteHaul experiences a significant degradation in performance when the system is subjected to interference. It is a concern that does WhiteHaul satisfy our need and requirements and prompt us to carry a research on the performance of WhiteHaul in aggregating multiple Wi-Fi channels.

1.2 Objectives

The main objective of the research is to find a highly efficient, inexpensive, and readily available multichannel Wi-Fi aggregation solution. We will examine an existing solution, WhiteHaul, for wireless aggregation and identify areas for improvement. We will also investigate the impact of interference between multiple Wi-Fi channels on WhiteHaul performance and find the root cause of the decrease in performance. We will focus on evaluating the effectiveness of different algorithms for managing the flow of data across multiple channels in WhiteHaul. Last but not least, this research will seek to improve WhiteHaul or suggest alternative solutions.

1.3 Contributions

The main contributions of the project can be summarised as follows:

1. Examine the performance of WhiteHaul under interference.
2. Examine whether changing the MPTCP scheduler has an effect on throughput.
3. Examine whether changing the MPTCP congestion control has an effect on throughput.
4. Demonstrate MPTCP congestion control algorithms play an important role in determining the maximum total aggregation throughput.
5. Demonstrate MPTCP schedulers play a vital role in goodput.
6. Discover the design flaw in WhiteHaul and provide adjustments.
7. Derive an upper bound for the aggregation throughput for any multipath aggregation system subject to wireless interference.
8. Describe a new variant of congestion control algorithm, TCP Edinburgh, which performs better than WhiteHaul.
9. Propose a list of potential improvements and discuss their limitations.

1.4 Report structure

Chapter 2 introduces key terminology and concepts that are essential for the report. WhiteHaul details are also included in this chapter.

Chapter 3 discribes the testbed we are carrying experiment on. The choice of different parameters is described and explained. It also includes the methodology for carrying

out the experiments in later chapters.

Chapter 4 includes the experimental result and discuss the observations. In section 4.1, we examine the performance of WhiteHaul as a baseline. We hypothesise the scheduler degrades the performance of WhiteHaul and disprove it in Section 4.2. In Section 4.3 we discovered that changing the congestion control algorithm of WhiteHaul leads to an improvement in the throughput. We explore the microscopic view of the congestion control of WhiteHaul and discover the flaws in Section 4.4.

Chapter 5 provides a list of potential improvements and discusses their benefits and limitations. Section 5.1 uses a mathematical relationship to derive an upper bound of the performance of MPTCP. Based on the relationship, we provide three solutions for increasing the upper bound in Section 5.2. Section 5.3 addresses the problems in the WhiteHaul mentioned in Chapter 4 and proposes alternative congestion control algorithms.

Chapter 2

Background

2.1 Wi-Fi

Wi-Fi is a wireless communication technology based on the IEEE 802.11 family of standards which specifies the physical layer (PHY) and the medium access control (MAC) layer for implementing wireless local area networks. The first IEEE 802.11 standard was published in 1997[19], and since then several other standards have been developed. The most recent versions are IEEE 802.11ac (Wi-Fi 5) [18] in 2014 and IEEE 802.11ax (Wi-Fi 6E) [20] in 2020. Each standard offers different levels of speed, range, and performance.

Wi-Fi technology uses short-range radios to transmit signals in the unlicensed spectrum at 2.4, 5 and 6 GHz. Wi-Fi channel width is the range of frequencies that a Wi-Fi signal occupies. The wider the channel width, the more data can be transmitted at once. There are only 3 nonoverlapping channels of 20MHz in 2.4GHz whereas there are over 24 nonoverlapping 20MHz channels and up to 12 40MHz in 5GHz. Moreover, there are many competitors for the 2.4GHz band such as Bluetooth, amateur radio, and microwave ovens, while 5GHz suffers less interference. Hence, there are more opportunities to exploit the 5GHz Wi-Fi band, so in this paper, we will focus on Wi-Fi aggregation on the 5GHz band using Wi-Fi 5.

All 802.11 standards implemented Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA)[19] where a node will detect whether the channel is clean by carrier sense before transmission. If a channel is detected as busy, the node will wait for the transmission to finish; otherwise, it will wait a short time and then start transmitting Wi-Fi frames. Each time a Wi-Fi device transmits a data frame, it must receive an acknowledgement from the receiver side. If the sender does not receive an acknowledgement for the data frame, it will retransmit the data a few times before giving up and allowing the upper-layer protocols to handle the lost event.

An important factor affecting radio link quality is Signal-to-Interference-plus-Noise Ratio (SINR), which is given by

$$\text{SINR} = \frac{P}{I+N} \tag{2.1}$$

where P is the power of the incoming signal of interest, I is the interference power of the other signals in the network and N is some noise term. The higher the SINR, the better the channel quality is. The Shannon-Hartley theorem states that the maximum channel capacity of a specified bandwidth in the presence of additive Gaussian white noise is governed by the following formula[44, 43]:

$$C = B\log_2\left(1+\frac{S}{N}\right) \tag{2.2}$$

where C is the maximum data transimission rate, B is the channel width of Wi-Fi and $\frac{S}{N}$ is the signal-to-noise ratio that we use SINR here.

There are mainly two kinds of interference, namely Adjacent Channel Interference (ACI) and Co-Channel Interference (CCI). ACI arises when extraneous power from a signal in a nearby channel disrupts transmission. This can happen when two channels are adjacent or partially overlapped. This will add interference and noise to SINR and degrade channel quality. CCI occurs when two different radio transmitters use the same channel simultaneously. As every wireless channel has a fixed capacity given by (2.2), the available capacity must be shared between two senders. Transmitting simultaneously will lead to retransmission and further reduce the data transfer rate.

Due to the instability nature of the wireless medium, rate adaption algorithms are essential to exploit the scarce wireless resources under unstable channel conditions. Modulation and Coding Scheme (MCS) is used in IEEE 802.11 networks to define the data rate and modulation scheme used to transmit data over the wireless channel. MCS index is a metric based on several parameters that affect the transmission data rate and reliability, including channel widths, number of antennas, coding rate and modulation scheme. The higher MCS will have a higher data transfer rate, but requires higher SINR for proper functioning.

2.2 UDP

The User Datagram Protocol (UDP) is a widely-used transportation layer protocol in computer networking. As a connectionless protocol, UDP provides a fast and efficient means of transmitting data over the network without the need to establish and maintain a connection. It is an unreliable protocol which does not guarantee in-order packet delivery and does not retransmit upon loss of packets. In our paper, UDP is used to generate Constant Bit Rate (CBR) traffic for interference.

2.3 TCP

Transmission Control Protocol (TCP) is a connection-orientated protocol that establishes a logical connection between two hosts and provides a reliable and ordered delivery of a stream of bytes over a network[10]. TCP works by fragmenting data into small packets and attaching a sequence number to each packet that allows the receiver to reassemble the packets in the correct order. To prevent the sender from overflowing the receiver buffer by sending too much data, TCP provides flow control by having the

sender maintain Receiver Window Size (RWND), the size of the free buffer space on the receiver side.

Congestion control

An important feature of TCP is the congestion control algorithms [4], which aim to prevent the network from being congested. TCP will maintain a Congestion Window Size (CWND) whose size is the number of bytes the sender can send in the network at any time. Ideally, CWND is equal to the Bandwidth-Delay Product (BDP), the product of a data link's capacity (in bits per second) and its round-trip delay time (in seconds). The data transmission rate is governed by $\min(\text{CWND}, \text{RWND})$.

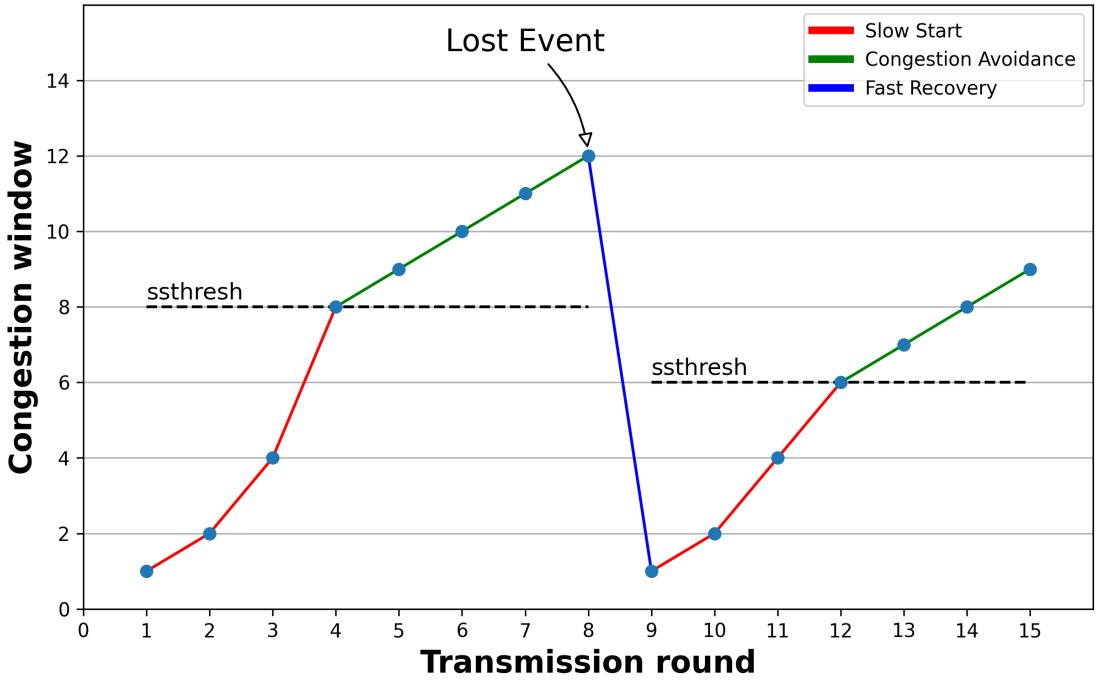


Figure 2.1: TCP Congestion Control using AIMD

A well-implemented congestion control algorithm consists of three components: slow start, congestion avoidance and fast recovery. Figure 2.1 shows how the congestion window changes during each transmission round. TCP uses the state variable Slow Start Threshold (SSTHRESH) to determine whether the slow start (CWND $<$ SSTHRESH) or congestion avoidance (CWND $>=$ SSTHRESH) algorithm should be used.

Slow Start At the initilisation of TCP connection or after a long idle period, TCP will enter slow start state. The value of CWND begins at 1 and increases by 1 every time a transmitted segment is first acknowledged. This process results in a double of the sending rate every RTT. Thus, the TCP send rate starts slow but grows exponentially during the slow start phase.

Congestion Avoidance The Additive-Increase/Multiplicative-Decrease (AIMD) algorithm is used by combing the linear growth of the congestion window when there is no congestion with an exponential reduction when congestion is detected. A

common implementation is to increase CWND by $1/\text{CWND}$ for every newly acknowledged packet. Until a lost event is observed, TCP will set SSTHRESH to $\text{CWND}/2$ and reset CWND to 1.

Fast Recovery When the sender receives three duplicated acks, loss of packet(s) is detected and TCP will enter the fast recovery state. The sender will quickly retransmit the missing packets and increase CWND by 1 for every acknowledgement of the missing packets until no duplicated acknowledgements are received.

TCP CUBIC is one of the variations of TCP congestion control algorithms which uses a cubic curve to increase CWND instead of a linear increase[16]. It is the current default congestion control algorithm in the Linux kernel[27]. TCP Illinois is another variant designed for long-fat networks[29].

2.4 MPTCP

Multipath Transmission Control Protocol (MPTCP) is an extension to tradition TCP that allows multiple paths between two endpoints to be used simultaneously[14].

In traditional TCP, a single path is used between a source and a destination, which can lead to suboptimal performance in certain network scenarios, such as when the path is congested or has high latency. MPTCP allows TCP to use multiple paths simultaneously to improve performance, resilience, and resource use. This is accomplished by splitting data into multiple subflows, each with a unique sequence number, and sending them over different paths. The subflows are then recombined at the receiving end to form the original data stream.

MPTCP operates in a transparent manner, which means that it does not require any changes to the underlying network infrastructure or the applications that use TCP. MPTCP is designed to fail back to traditional TCP when the network does not support MPTCP[21].

From Figure 2.2, we can see that MPTCP composes of three main components

- Path Management: This component is responsible for detecting and selecting multiple paths between the endpoints and managing the data transfer over these paths. It includes path discovery, path establishment, path selection, and path monitoring.
- Schedulers: This component is responsible for dividing the data stream into segments and deciding how to distribute these segments over the available paths, taking into account the characteristics of each path and the level of congestion[34]. It includes segment size selection, segment scheduling, and retransmission strategies. Head-of-Line (HoL) block is a phenonmmenon that can occur in MPTCP when a data segment is lost or delayed on one path, causing subsequent segments to be blocked in the receiver buffer until the missing packet is received. The key responsibility of schedulers is to reduce HoL[32].
- Congestion Control: This component is responsible for managing the sending rate over the available paths to avoid congestion and optimise the utilisation

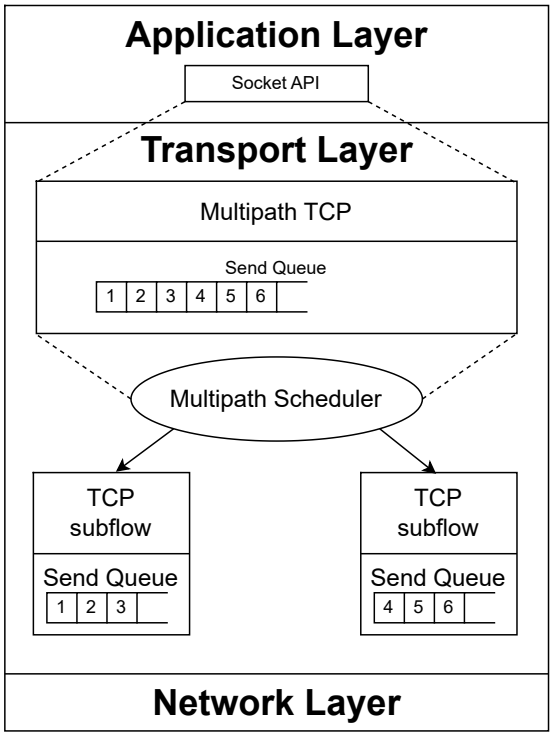


Figure 2.2: MPTCP protocol stack

of network resources. Each TCP subflow has its own CWND and there is an MPTCP-level CWND.

These three components work together to ensure that MPTCP can efficiently use available network resources, avoid congestion collapse, and maintain the stability and fairness of the network.

A practical multipath congestion control algorithm must have the following three desirable properties[38]:

Goal 1 Improve Throughput: A multipath flow should perform at least as well as the best available single-path flow.

Goal 2 Do no harm: A multipath flow should not take more than would be obtained by a single path TCP in shared resources.

Goal 3 Balance Congestion: A multipath flow should move as much traffic as possible away from its most congested paths without violating the first two goals.

2.5 WhiteHaul

Taking advantage of the white space in unlicensed television, WhiteHaul aggregates multiple chunks of TV white space with MPTCP as a link-level tunnel abstraction and cross-layer congestion control to manage network traffic[23]. It can achieve 1Gbps throughput of a distance of more than 10km with 99.99% availability[17].

For the hardware layer, WhiteHaul consists of a TV white space conversion substrate to convert the Wi-Fi signal generated by each radio card to the TV band, and a power splitter to combine the converted singals into one for transmitting over TV white space. It also has a LoRa[30] interface for coordination between two WhiteHaul nodes.

For the software layer, WhiteHaul consists of three modules.

1. (Coordination Module) This module is responsible for sending control messages between two WhiteHaul nodes using LoRa.
2. (Interface Configuration Module) This module is responsible for sensing the channel quality to deduce which frequency ranges are available for use. It will also obtain a list of approved frequency ranges and maximum allowed transmission power from a geolocational database.
3. (Traffic Management Module) This module is responsible for fairly allocating time slots for forward and reverse traffic, as well as efficiently scheduling the traffic among available subflows using MPTCP.

WhiteHaul's congestion control algorithm is an uncoupled MPTCP congestion control algorithm based on TCP Illinois, with customised modifications to achieve highthrough-put for long-range wireless network.

In our research, we will use a stripped-down version of WhiteHaul as described in 3.

2.6 Related Work

Since the emergence of MPTCP, many research studies have explored the possibilities of using MPTCP under different network conditions. Most of the work is done on Wi-Fi and LTE[7, 1, 12, 31, 8] or on the entire ethernet scenario in data centers[40, 41, 9]. Some work is done on handover between heterogeneous networks[35]. However, very few articles have explored the subject of MPTCP in simple Wi-Fi scenarios[36, 47], especially investigating the behaviour of MPTCP under Wi-Fi interference[48].

Chapter 3

Methodology

To investigate the performance of MPTCP in Wi-Fi settings with and without interference, we performed a series of experiments to measure its behaviour in various wireless environments and configurations. This section describes how the experiment is organised and performed.

Section 3.1 explains how the project and experiments are managed. The hardware and software aspects of the experimental setup are described in Section 3.2. Section 3.3 lists out the important configurations and gives a brief reason for the considerations. Section 3.4 shows how each experiment is carried out and what performance metrics are used. Section 3.5 concludes our necessary precautions to increase the reliability of the result.

3.1 Project Management Strategies

We begin by measuring the throughput of WhiteHaul in the clean channel case and examine its behaviour in the presence of interference, aiming at reproducing the field findings that the performance of WhiteHaul degrades significantly when there is near-by interference. The baseline result will motivate us to investigate how other MPTCP configurations such as different congestion control algorithms and schedulers perform compared to WhiteHaul.

Due to the limited time available to complete the project, we decided to conduct a small set of preliminary experiments that can represent a wide range of test cases and provide insights into areas that were most interesting to explore. These initial tests were helpful in gaining a better understanding of the data collected; then, we can fine-tune the test setups and measurement metrics.

Given the nature of the experiments, most of our time was utilised in conducting tests, debugging and fixing the flaws in both hardware and software. Additionally, we have regular meetings with our supervisor in order to review our progress, discuss our interpretation of the results, and determine the path forward for future experiments.

3.2 Experiment Setups

To stay in line with our research's objective, we use a simplified version of WhiteHaul. We remove the use of the TV conversion substrate and directly use Wi-Fi for the experiment. We also remove the power splitter and use a separate Wi-Fi antenna for each Wi-Fi frequency used. We do not have LoRa for the control channel as we want all the communication to be carried out in the 5GHz Wi-Fi band.

Hardware and Software

Our experimental setup consists of two Intel® NUC 11 Essential Kit - NUC11ATKC4 (Intel(R) Celeron(R) N4505 @ 2.00GHz, 8GB of RAM) that run the software layer of two WhiteHaul nodes. Ubuntu Focal 20.04 Server is installed on both machines with the modified MPTCPv0.96 Linux Kernel 5.4 implementation, which has the full support of MPTCPv1[14]. Each machine is connected to two radio cards through Gigabit Ethernet (GbE) interfaces. The radio cards are Mikrotik RouterBoard - RB922UAGS-5HPacD, running RouterOS 6.48.6 Long-term. As the NUC have only one GbE interface, an additional USB 3.0 to Gigabit Ethernet Adapter NIC is used.

We use another two machines with another two sets of radio cards to generate a controlled interference source. The UDP client is placed close to the WH-2 of the server side, in order to generate sufficient interference to the receiver. We use RF Explorer 6G Combo handheld digital spectrum analyzer to check the quality of the channel.

Physical Environment

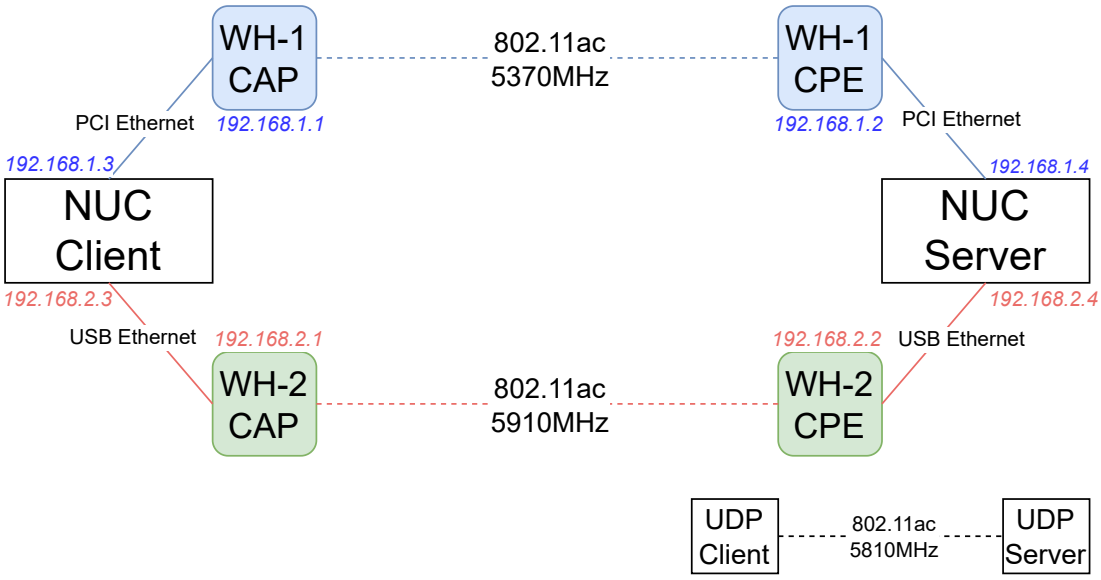


Figure 3.1: Physical Setup

Furthermore, although all devices run at different frequencies, to avoid possible interference between these electronic devices, we use Ethernet cables to extend radio cards.

The two WhiteHaul nodes that make up the end points of the link under test are placed in the same laboratory two to three metres apart.

Using the spectrum analyser, we discover that the Wi-Fi network suffers from significant nearby interference caused mainly by many IEEE802.11abgn access points point operating in the common non-overlapping channels in the 5GHz band. Most of them are university networks or experiments done by other groups in the network lab. Therefore, we searched the white space in the 5GHz spectrum and found that U-NII-2B 5350-5470 MHz, U-NII-4 5850-5950 MHz and U-NII-5 5925-6425 MHz are clean, as they are not for standard Wi-Fi usage. We put the first link WH-1 at the frequency range 5910-5990MHz as a clean link as there is little or no usage in this particular frequency. The second link WH-2 is put at the frequency range 5370-5450MHz and the controlled UDP traffic will use the frequency range 5260-5340MHz, which is chosen to stay close to WH-2 but far away from WH-1.

3.3 Experiment Configuration

Connection Parameters

Linux uses a default initial SSTHRESH of infinity (*TCP_INFINITE_SSTHRESH*) when initialising TCP sockets and caches parameters for the per-destination TCP connection[42]. It would be undesirable to make a fair comparison between tests, so we disable "TCP metrics save" to prevent SSTHRESH from being reused.

We keep Linux's default initial window size of 10 packets, enable TCP timestamps for better RTT measurement [5], and TCP Selective Acknowledgement (SACK) to reduce the number of retransmissions[37].

Memory Allocation

As MPTCP uses multiple subflows, a larger buffer size is required to handle more packets, probably out of order, from different paths. Related work shows a decrease in performance if the allocated buffer is too small[39, 49]. Therefore, we set the maximum sent buffer size (*wmem*) and the received buffer size (*rmem*) of the kernel to 16 MiB for queues in all protocols.

Wi-Fi configuration

As the Mikrotik RouterBoard support Wi-Fi 5 IEEE 802.11ac and we are only interested in the 5GHz band, we set it to "5ghz-onlyac" to prevent it from dynamically switching to other modes. For each RouterBoard, it is configured to bridge the Ethernet port and WLAN port so that the NUC devices can connect to the radio card using Ethernet and yet send data through the Wi-Fi spectrum. For each pair of RouterBoards on the client and server (two pairs in total), they are configured to form a link as a Point-to-Point (PTP) wireless bridge with names "WH-1" and "WH-2" respectively.

"WH-1" and "WH-2" use the subnet "192.168.1.0/24" and "192.168.2.0/24" respectively.

Each NUC uses “fullmesh” as the MPTCP path manager, and configures routing tables by “iproute2” to prevent packets from being sent to the wrong subnet.

3.4 Experiment Methodology

Unless stated otherwise, all experiments for ordinary TCP measurement use Linux’s default congestion control CUBIC as a baseline. All the traffic directions originate from the NUC Client (sender) to the NUC server (receiver).

The Mikrotik RouterBoard supports 20MHz, 40MHz, and 80MHz, and thus different bandwidths. We will change the channel width of each wireless link to investigate the MPTCP behaviour for different bandwidth combinations. We will use the compact notation (a-b) to indicate the channel width configuration for a particular experiment. (20-80) means WH-1 set to 20MHz whilst WH-2 set to 80MHz.

Three important pieces of software are used in the experiments. *iPerf* is used to generate traffic and measure the throughput of the network. *tcpdump* is used to capture traffic at a specified interface in *pcap* for further analysis. *tcptrace* is used for calculating the throughput of MPTCP contributed by each subflow.

In the report, we have three types of tests:

1. (*Individual*) WH-1 running *iPerf* while WH-2 idle and vice versa. Concurrent transmissions are not made and ordinary *tcp* is used.
2. (Simultaneous) Both WH-1 and WH-2 run *iPerf* with ordinary TCP. Both subflows are benchmarked concurrently.
3. (MPTCP) MPTCP will use both WH-1 and WH-2.

When studying the interference cases, the UDP client will use *iPerf* to generate CBR traffic of 20 Mbps. As the UDP client is placed close to the NUC server, it will increase the interference power on the receiver side and thus reduce SINR.

The following performance metrics are used for comparing the result.

Throughput Throughput is the direct feedback of an aggregation solution. To have a fair comparison, we need to normalise the result. Defining a multipath aggregation scenario S as a set of capacities $C_i > 0 (i \in 1, ..., n)$, representing the average performance of n individual transmission paths, and a measured throughput x (with $0 \leqslant x \leqslant \sum_i C_i$) achieved by a solution. An intuitive way of normalisation is the aggregation ratio, which is calculated by

$$A_r = \frac{x}{\sum_i C_i} \tag{3.1}$$

However, one goal of multipath aggregation is that the performance should be better than using a single path, and we need a better metric to reflect this goal. We decide to take 0 as the solution performs the same as the best single path performance, -1 as 0 Mbps and 1 as perfect aggregation[22]. The metric should be

monotonic, preserve a partial order relationship, and preferably linear for easier comparison. The aggregate benefit is defined as follows.

$$A_b = \begin{cases} \frac{x - \max_i C_i}{(\sum_i C_i) - (\max_i C_i)} & x \geqslant \max_i C_i \\ \frac{x - \max_i C_i}{(\max_i C_i)} & x < \max_i C_i \end{cases} \tag{3.2}$$

The interpretation of A_b is how much more the solution can aggregate apart from the best single path. Notice that we cannot directly compare A_b of different signs due to the non-linearity at 0. Both the aggregation ratio A_r and the ggregation benefit A_b will be used for the comparison of the results in Sections 4.1 and 4.2.

TCP status To gain an in-depth understanding of how MPTCP reacts upon incoming events, it is necessary to access the kernel's internal status, specifically the TCP sockets for each MPTCP subflow. Traditionally, two methods are used: capturing *pcap* files to estimate values or modifying the kernel source code to print debug information. However, the former can yield inaccurate or even misleading results as the kernel may drop network packets before they can be captured by software. The latter method adds significant overead to the kernel and is prone to kernel panic. Instead, state-of-the-art eBPF technology is used, allowing custom code to be safely and efficiently executed without modifying the kernel itself. A C program is written to hook the *tcp_rcv_established* kernel function and extract useful information from *struct sock*, such as *snd_ssthresh* (SSTHRESH), *snd_cwnd* (CWND), *snd_wnd*, *srtt_us* (sRTT). This information will be used for the analysis described in Section 4.4.

3.5 Reliability of Wireless Experiments

Because Wi-Fi is highly sensitive to timing and the surrounding environment, we have to perform repeated trials of particular combinations to ensure consistent results. We will run five iterations for a given setting, with each run taking a minimum of 200 seconds for both Wi-Fi and TCP to stabilise. We allow a 10 second interval between consecutive trials to ensure that the kernel garbage collect the socker buffer, preventing the preceding experiment from influencing the next experiment.

Chapter 4

Results and Discussion

4.1 Initial Experiment

A wireless network is highly susceptible to the surrounding environment and nearby interference. An aggregation system needs to be resilient to interference and achieve sensible aggregation throughput.

To get an idea of how interference impact WhiteHaul's performance, we carried out an experiment with different combinations of channel widths and varying the interference. We recorded the throughput for both the clean channel and the channel with interference and compared the output to the individual TCP in the clean channel.

Channel Quality	Type	WH-1 (Mbps)	WH-2 (Mbps)
Clean	Individual	61.9	256.0
Clean	MPTCP	58.6	234.4
Interference	MPTCP	58.1	182.1

Table 4.1: (20-80) MPTCP WhiteHaul minRTT average throughput

Table 4.1 gives the result of (20-80) configuration. When the channel is clean, the aggregation ratio of this combination is $\frac{58.6+234.4}{61.9+256.0} = 0.92$ and the aggregation benefit is $\frac{58.6+234.4-256.0}{61.9+256.0-256.0} = 0.59$. However, in the presence of interference, the aggregation ratio of this combination is $\frac{58.1+182.1}{61.9+256.0} = 0.75$ and the aggregation benefit is $\frac{58.1+182.1-256.0}{61.9+256.0-256.0} = -0.06$. A negative aggregation benefit indicates that the aggregation solution performs worse than using a single path. This match with [23] field experience that the "efficiency" of WhiteHaul under interference degrades significantly compared to the clean channel situation. A summary of different channel width combinations is shown in Figure 4.1 and Figure 4.2.

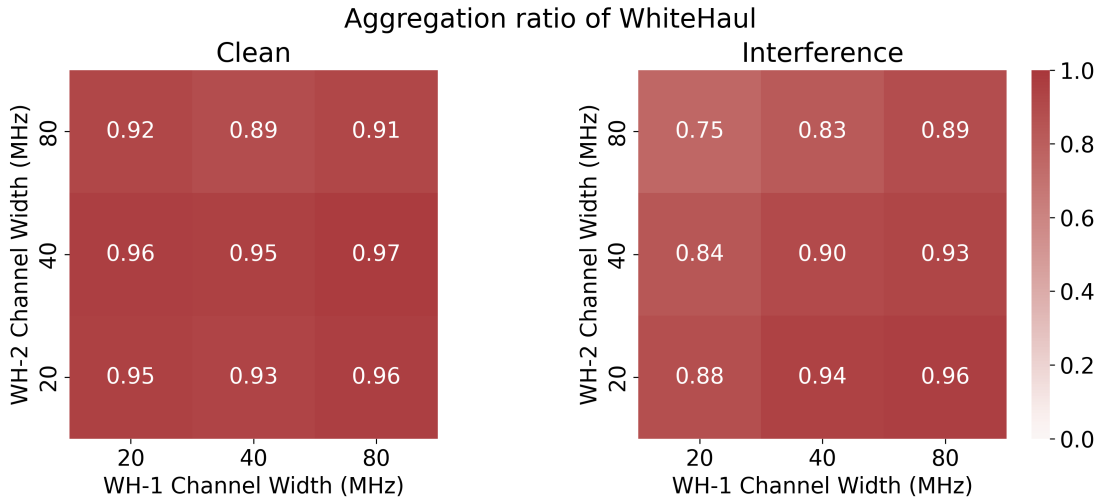


Figure 4.1: WhiteHaul aggregation ratio with different combinations of channel widths

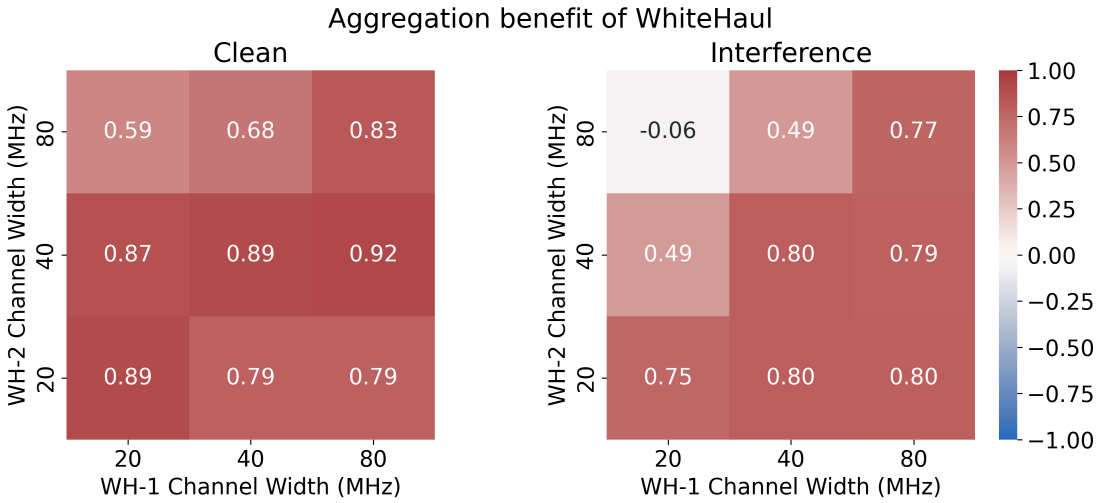


Figure 4.2: WhiteHaul aggregation benefit with different combinations of channel widths

A clear observation is that when WH-2 (the link subject to controlled interference) has a channel width wider than that of WH-1, the aggregation performance decreases. The greater the difference between the channel width of the two links, the poorer the performance. We hypothesise that it is the scheduler issue that the scheduler used in WhiteHaul is unable to handle the heterogeneity in RTT and latency. Wi-Fi networks under interference will have many retransmissions at the layer 2 protocol and very few reported back to the high layer such as TCP as a loss event. It mainly increases the RTT and results in larger latency, making the network exhibit heterogeneity. [12] shows path heterogeneity can hinder the aggregation throughput by MPTCP, mostly due to the HoL-blocking which causes higher receiver memory usage and subflow bandwidth underutilisation. In the next section, we are going to test our hypothesis by reusing the congestion control in WhiteHaul but with different schedulers.

4.2 Is scheduler the culprit?

To verify our hypothesis made in the previous section, we investigate how the choice of schedulers affects the performance of WhiteHaul. There are several MPTCP schedulers designed, including BLEST[13], ECF[25] and QAware[46]. BLEST reduces HoL by considering the size of the sender's congestion window and trying to minimise out-of-order packet delivery. On the other hand, ECF uses the number of remaining packets in the sender's buffer and estimates the completion time for the packet if it is sent on a slower subflow. QAware addresses the problem by utilising lower-layer information, the hardware queue occupancy, and is able to swiftly react to changes in network conditions. BLEST and ECF are implemented in the Linux kernel and the source code of QAWARE is available on GitHub. However, since radio cards act as an isolated device instead of a network interface, there is no method for the Linux kernel to report the hardware information of the radio cards to the QAware algorithm. We decided not to include QAware for comparison and used BLEST and ECF instead.

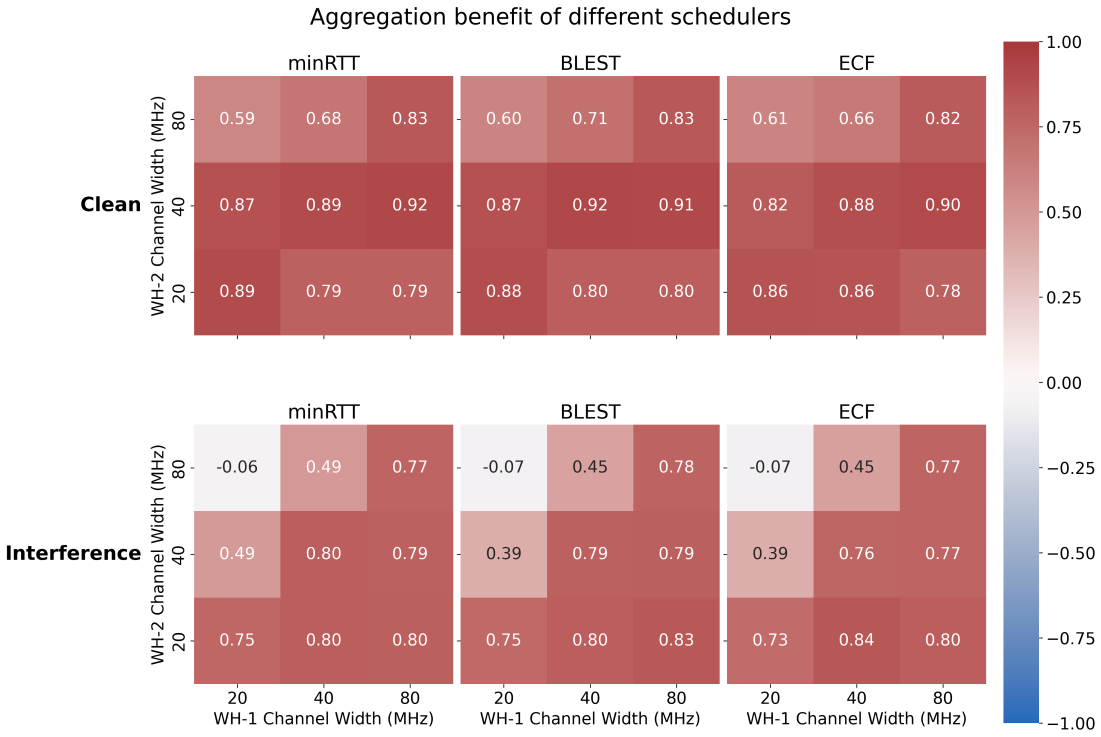


Figure 4.3: Aggregation benefit with different schdulers

We re-run the experiment by replacing the scheduler (minRTT) in WhiteHaul with BLEST and minRTT. We use aggregation benefit as a metric for a fair comparison. From Figure 4.3, there is no observable significant difference in performance between the schedulers. The slight difference in number can be regarded as random noises due to the unstable nature of the wireless network. Looking at the aggregation benefit from average total throughput may not be insightful about how the schedulers behave. A more useful visualisation is a breakdown analysis of MPTCP throughput and observing how much bandwidth a scheduler allocates to each subflow. We use *tcpdump* to capture

the TCP traffic for each interface into *pcap* files and use *tcptrace* to calculate the actual throughput.

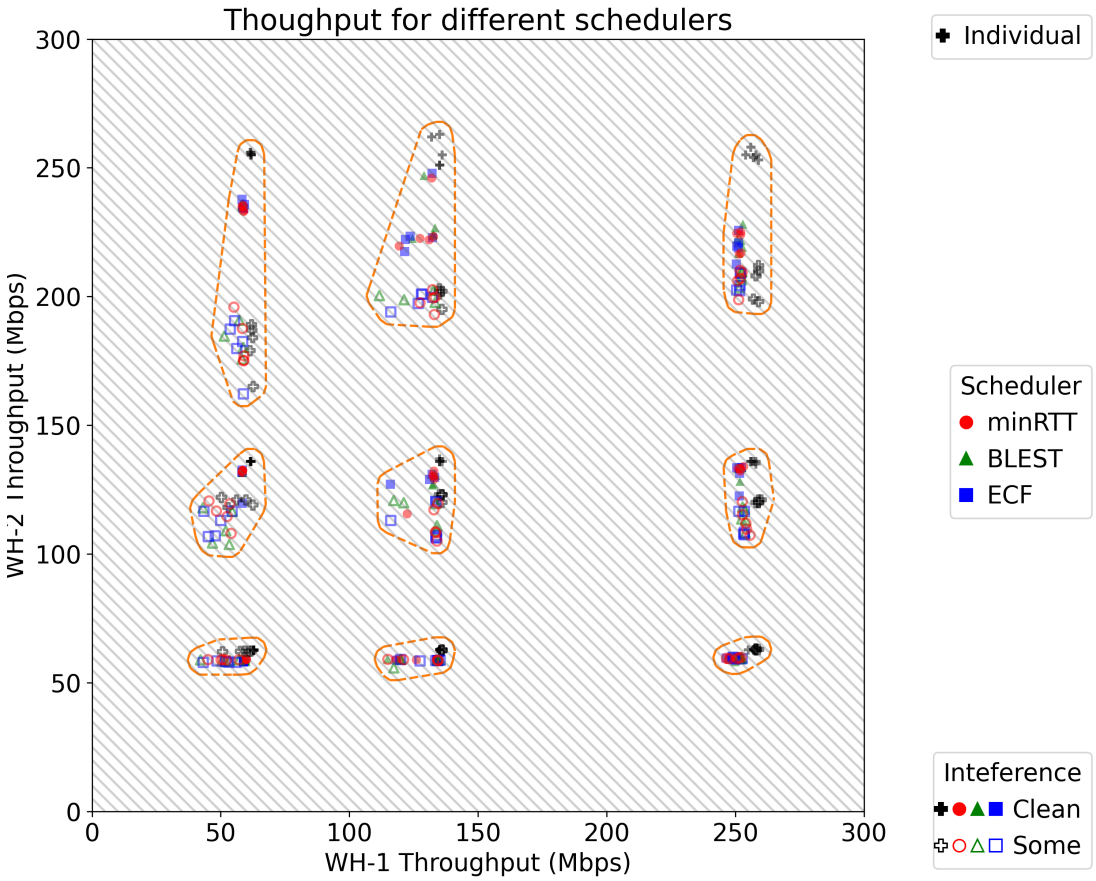


Figure 4.4: An overview of different schedulers for each channel width pair

To make the visualisation more readable, we use different colours (red, green, and blue) to represent different schedulers (minRTT, BLEST, and ECF, respectively). The capacity of each individual subflow is coloured black. We use solid markers to represent the clean channel case and hollow markers to indicate the presence of interference. Each orange cluster belongs to the same channel width pairs, as there are nine different channel width combinations for two subflows. We have included slanted grey lines with slope -1 for easier comparison. Any two points laying on the same grey line aggregate to the same total throughput. The total throughput for any two adjacent grey lines differs by 10Mbps. The closer the point is to the top right corner, the higher the aggregated throughput. The closer the point is to the bottom left corner, the more bandwidth is allocated to subflow WH-1, and vice versa, top right corner for subflow WH-2.

Figure 4.4 gives an overview of the result. When there is interference, we can see a small variation in the throughput of WH-1 which is free from interference. Conversely, there is a large variation in the throughput of WH-2 which is affected by the controlled UDP interference. The larger the channel width of WH-2, the greater the variation in the experiment result. This can be attributed to the fact that the wider channel width is more susceptible to interference. Greater interference will result in transmission failure

and more retransmissions in the layer 2 Wi-Fi protocol, and thus variable latency. It makes schedulers' job hard to adapt to rapid changes in latency.

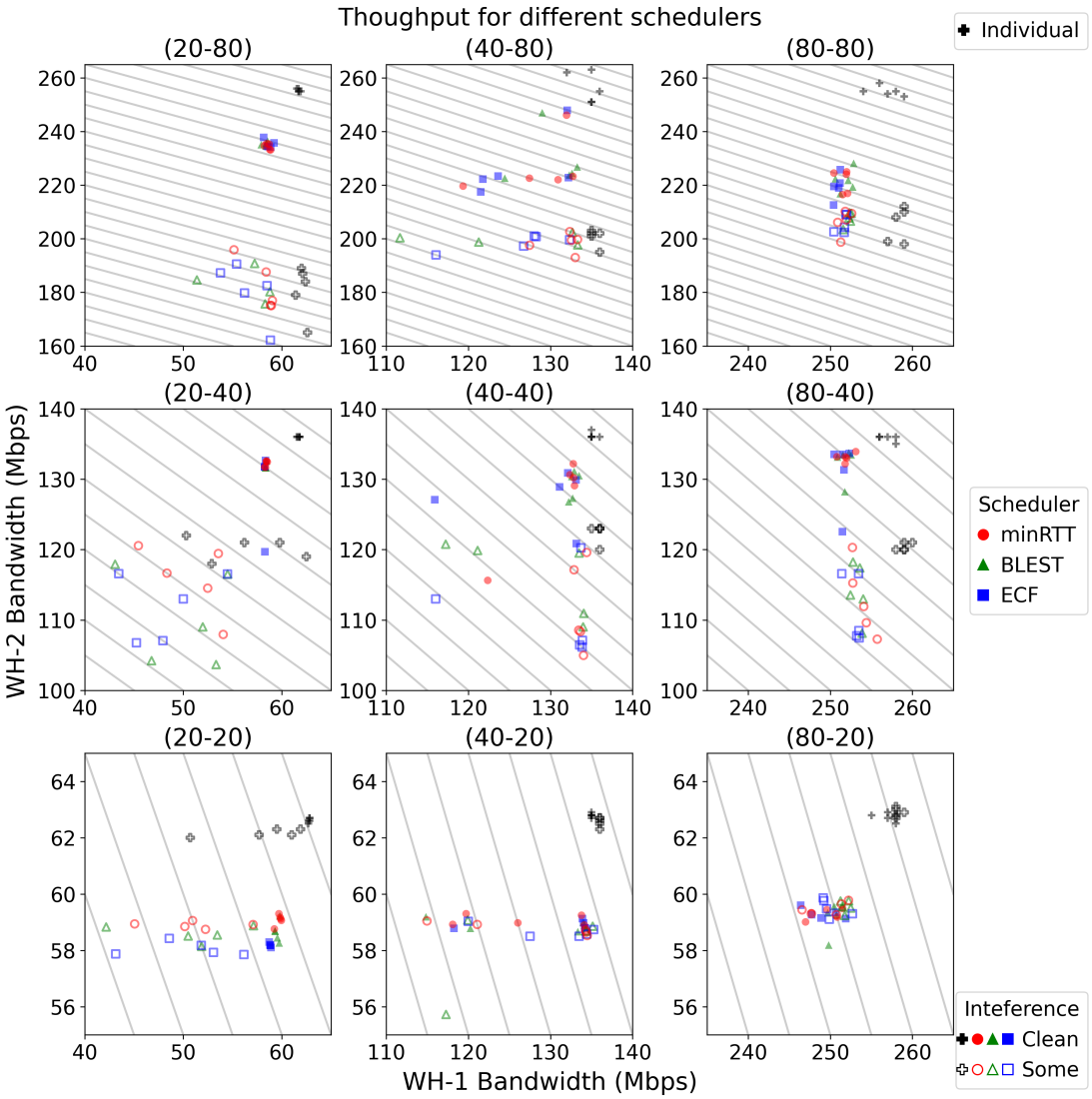


Figure 4.5: Detail of different schedulers for each channel width pair

Figure 4.5 provides more detail within the same channel width configuration. We can observe that in a clean channel case, all schedulers perform similarly. For the interference case, the result of the three schedulers scatters uniformly, making no significant difference.

Hence, we safely conclude that schedulers are not the main culprit of the poor performance of the aggregation throughput by WhiteHaul.

4.3 Is congestion control the culprit?

In the previous section, we rejected the hypothesis that schedulers degrade the performance of WhiteHaul under interference. We turn our focus on the other component of

WhiteHaul - the congestion control algorithm. WhiteHaul is designed with a customised congestion control algorithm to target backhaul traffic with a stable link. There could be a chance that the congestion control algorithm in WhiteHaul acts poorly in the wireless network, causing the schedulers unable to fully utilise the available capacity of the subflow. Since the schedulers take the sender's congestion windows into account when scheduling the packets, if the congestion control algorithms shrink the windows aggressively upon the change in network conditions, the total throughput is indeed mainly affected by the reduced window size. This reasoning prompts us to investigate whether using other congestion control algorithms could lead to a better result.

There are many available congestion control algorithms. Linux default congestion control algorithm CUBIC can be used as an uncoupled congestion algorithm for MPTCP while LIA is a standardised coupled congestion algorithm[38]. Similarly to what we have done in Section 4.2, keeping the scheduler as minRTT, we replace the congestion algorithms in WhiteHaul with CUBIC and LIA, and evaluate the aggregation benefit for different combinations of channel widths.

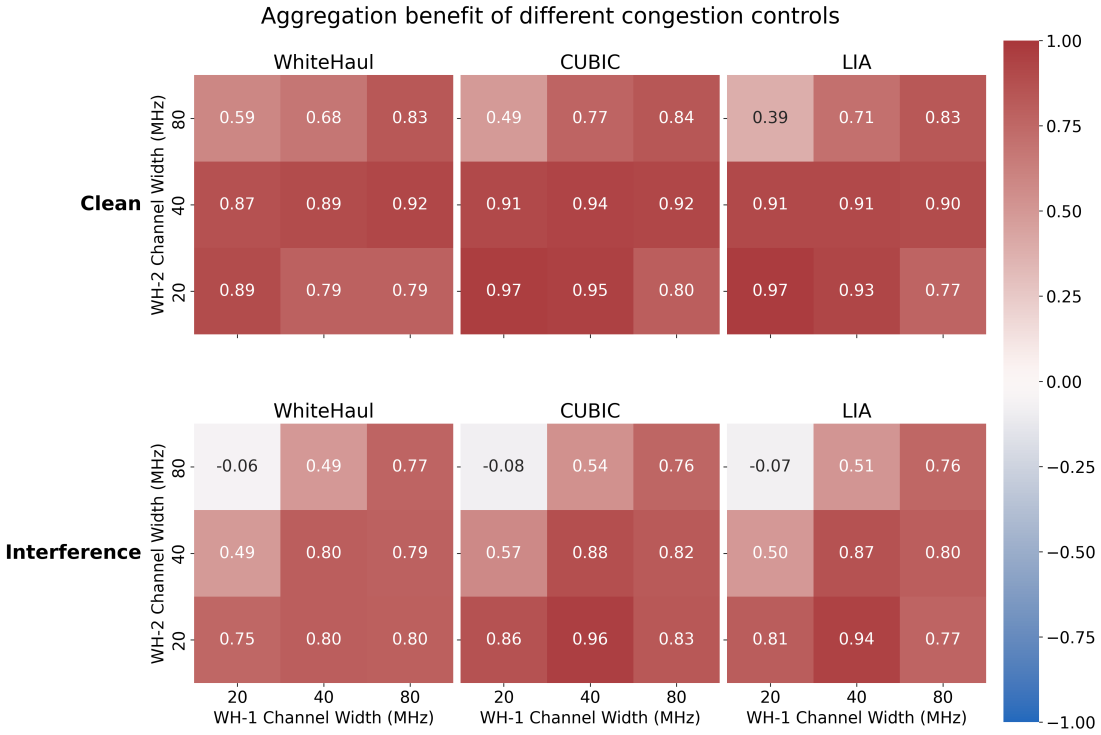


Figure 4.6: Aggregation benefit with different congestion controls

In Figure 4.6, all congestion controls show similar patterns when we add interference, as we discovered in Section 4.2. (20-80) still performs the worst regardless of the congestion control algorithms used, and the aggregation benefit is low when the channel width of WH-2 is greater than that of WH-1. However, we can see a significant difference in the performance of different congestion control.

1. LIA performs poorly compared to WhiteHaul and CUBIC. This can be explained by the fact that LIA as a coupled congestion control algorithm uses the same

congestion control algorithm for all subflows. When increasing the congestion window of one subflow, it will take other subflows' characteristics into account. It ensures that each path receives a fair share of the available bandwidth and acts with friendlyness on the network bottleneck. Conversely, WhiteHaul and CUBIC are both uncoupled congestion control algorithms where a flow could decide its CWND without considering other flows CWND. It results in aggressively utilising the available bandwidth in each individual subflow, and hence in larger overall throughput.

2. For most of the time, CUBIC performs better than WhiteHaul, in both clean channel cases and the presence of interference. There is no trivial explanation based on the numbers displayed and the working principle of the congestion control algorithms. This finding leads us to plot the performance graph in Figure 4.7 to explore more properties of each congestion control algorithm.

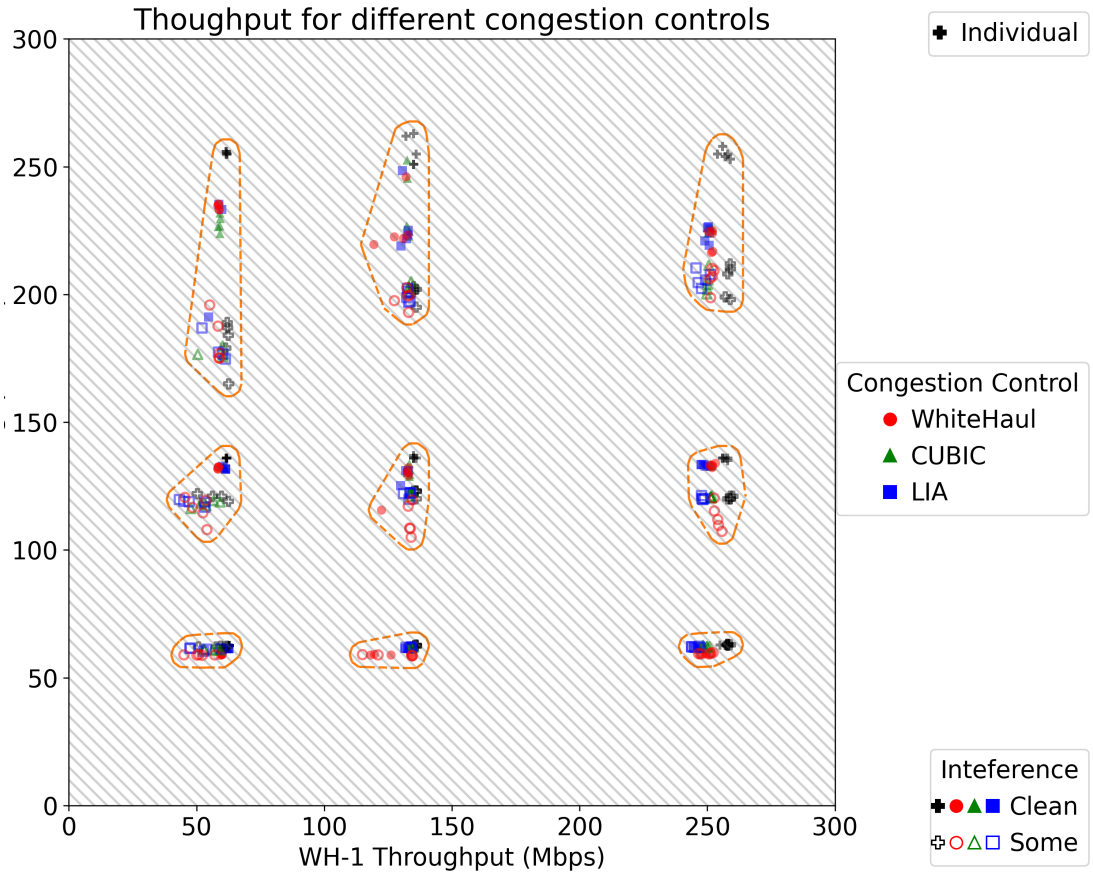


Figure 4.7: An overview of different congestion controls for each channel width pair

Figure 4.7 and 4.8 summarise the performance of different congestion control algorithms. Although there are no notable differences among congestion control algorithms, WhiteHaul has a large standard deviation in throughput when there is interference. When the channel width of WH-2 is 20MHz, we can see that the throughput of WH-2 is consistently lower than other congestion control algorithms with a wide variation in WH-1 throughput. When the channel width of WH2- is 40MHz or 80MHz, the

variation in WH-1 throughput is small but large in WH-2 throughput. This means that WhiteHaul's behaviour is unstable in case of interference where there are non-negligible fluctuations in RTT and delays. We declare that the congestion control algorithm is one of the important factors that affect the aggregation ability of MPTCP What is causing this behaviour requires a more detailed analysis of CWND and other variables affected over time. With this question in mind, we will have a microscopic view of the congestion control algorithm in the next chapter.

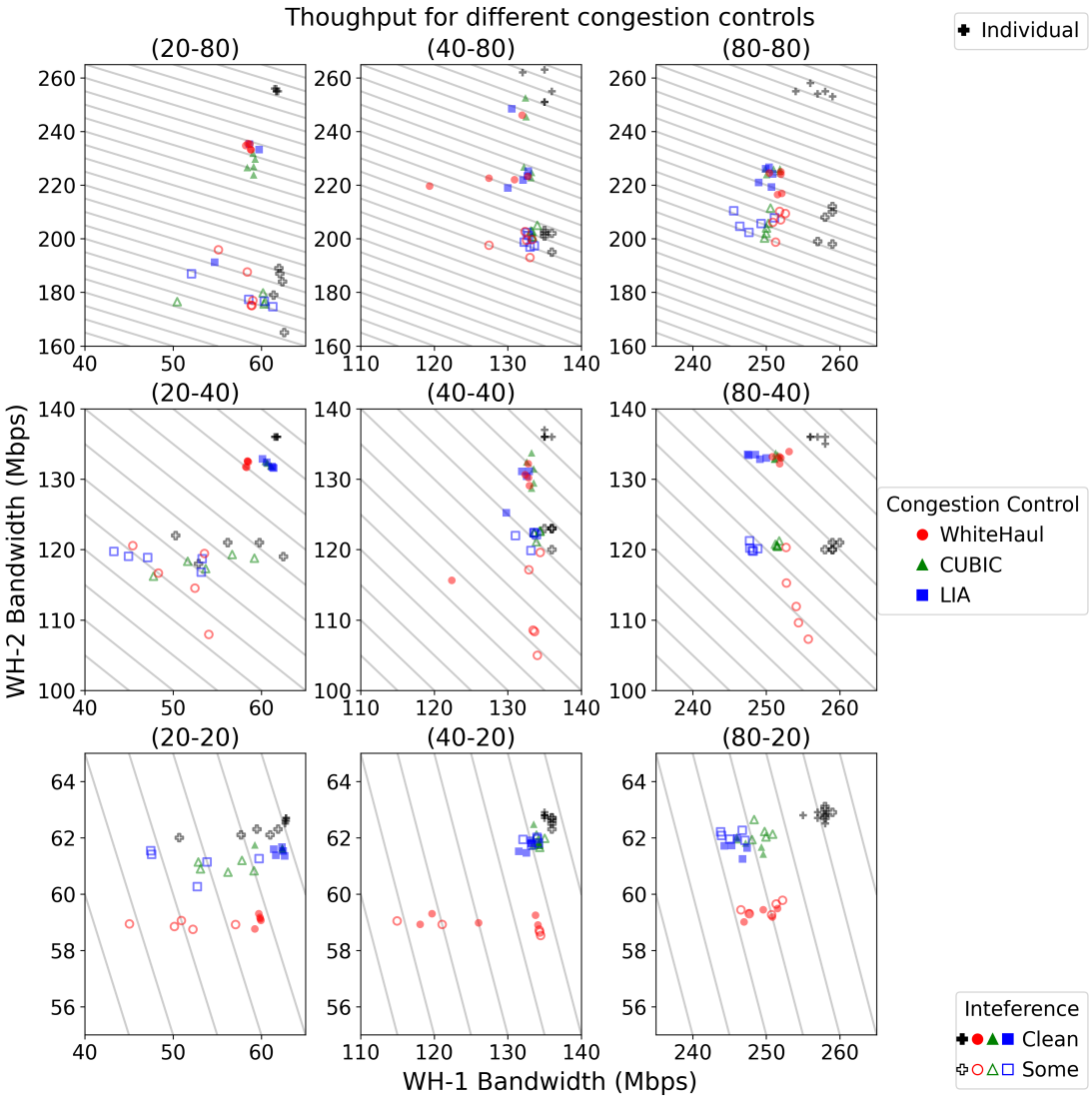


Figure 4.8: Detail of different congestion controls for each channel width pair

4.4 Microscopic view of congestion control

To track the status of WhiteHual over time, we need to know the details of every TCP socket sent and received. More important we need to track the SSTHRESH, CWND and sRTT to have an idea of how the congestion control algorithm responds to a change in network events. We use eBPF to hook the kernel function *tcp_rcv_established* and

retrieved relevant information for each incoming TCP packet from each subflow. We keep tracing the sender's SSTHRESH, CWND and sRTT over time and summarise the data into plots. We repeat the experiment with different congestion control algorithms. The results are summarised in Figure 4.9 and Figure 4.10.

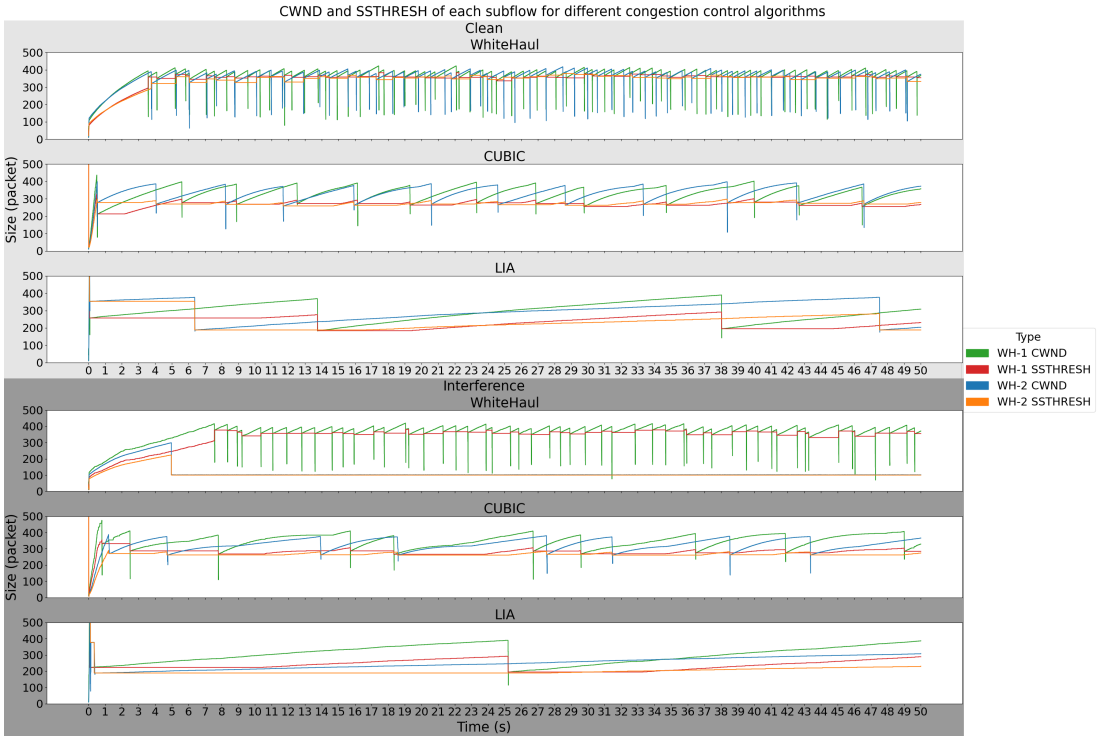


Figure 4.9: CWND and SSTHRESH of each subflow for different congestion control algorithms.

First of all, we discover the slow convergence of WHiteHaul when a TCP connection is initiated. While CUBIC and LIA take less than 1 second to reach the expected CWND, WhiteHaul takes 4 seconds for a clean channel and 7 seconds in the presence of interference. The slow start of WhiteHaul ended too soon, leading to a slower increase to the expected congestion window size during the congestion avoidance state. Secondly, we observed that when there is interference, the CWND and SSTHRESH are clamped at 100 upon a packet's sRTT exceeding 0.05 seconds. In Figure 4.10, we see a peak of sRTT at 5 seconds followed by a drop of CWND and SSTHRESH to 100 in Figure 4.9. We need concrete implementation details of WhiteHaul to explain the above phenomenon.

From the WhiteHaul source code, we discover that the initial SSTHRESH is set to 20 (line 111 A.1). This explains why slow start ends quickly since it only takes 5 RTT ($2^5 = 32 > 20$) for the TCP connection to exit the slow start state and enter the slow increase state of the congestion avoidance state. In addition, a default and fixed value of 100 is set for the advertised window size of each subflow (line 49 A.1), which means that the sender congestion window will always reduce to 100 when there is a loss event. Although it can be configured when the kernel module is loaded, it is impractical to change the value during transmission. Also, due to the instability nature of wireless networking, it is not sensible to set an arbitrary value for

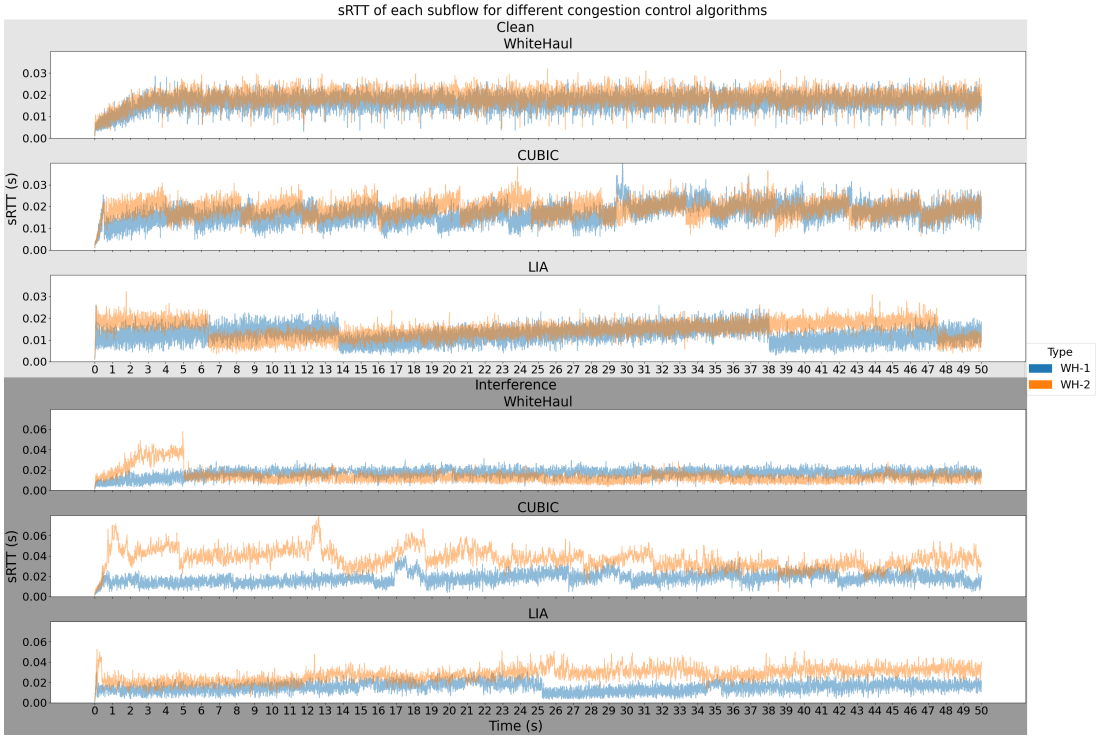


Figure 4.10: sRTT of each subflow for different congestion control algorithms

the advertised window size suitable for any interference cases and network conditions. On the other hand, the queueing budget delay used in WhiteHaul is hard-cored at 50000 microseconds(*qDelayBudget_us* at line 228 A.1), that is, 0.05 seconds, which coincides with the observation in Figure 4.10. It suffers from the aforementioned problem of not having flexibility to handle different network conditions.

After reviewing the logical flow of the implementation of the WhiteHaul congestion control, we found that some variables are not properly reset after use (line 225 of the function *alpha* A.1). Consequently, when there is a huge spike in RTT exceeding the queueing budget delay, *rtt_above* is set to true and the window clamping code will set both CWND and SSTHRESH to the advertise window size of 100. However, when RTT drops below *qDelayBudget_us*, *rtt_above* does not reset, the window clamping condition always holds and therefore cannot increase CWND. It explains the trend of CWND and SSTHRESH in Figures 4.9 and 4.10.

To confirm our argument that small SSTHRESH slightly worsens the performance of WhiteHaul, we conducted an experiment with small file transfer that can be transferred in around 10 seconds. We use the (80-80) channel width configuration, minRTT as the scheduler with different congestion control algorithms, and repeat the test 20 times. The result in Figure 4.11 shows that all congestion control algorithms perform similarly clean channel cases, but WhiteHaul takes longer to download small files than CUBIC when there is interference. It confirms our argument that the small SSTHRESH slightly worsens the performance of WhiteHaul.

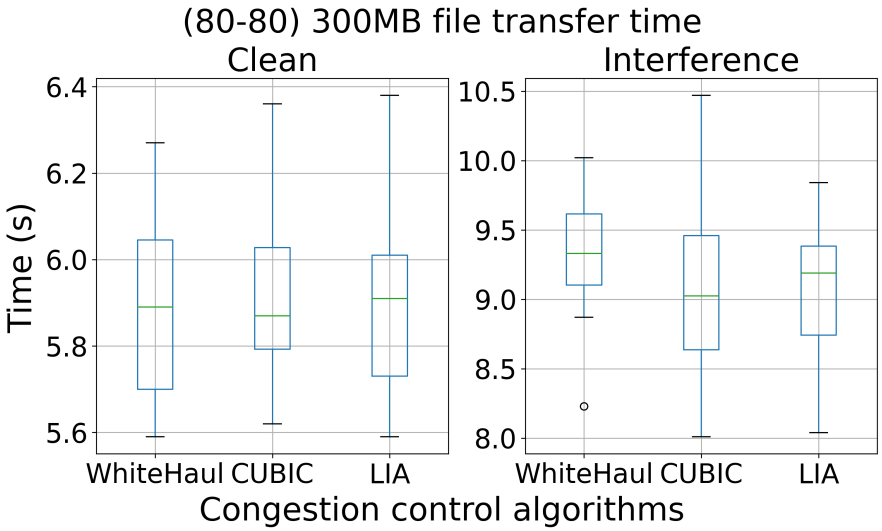


Figure 4.11: Time taken to transfer 300MB file using different congestion control

Chapter 5

Proposed Improvements

5.1 Upper bound of the throughput of MPTCP

Simultaneous TCP

Channel Quality	Type	WH-1 (Mbps)	WH-2 (Mbps)
Clean	Individual	229	261
Clean	Simultaneous	149	258
Clean	MPTCP	148	248
Interference	Individual	215	112
Interference	Simultaneous	164	101
Interference	MPTCP	177	71

Table 5.1: (80-80) MPTCP WhiteHaul minRTT average throughput

A natural question that pops into the mind is why throughput is low for both links when using MPTCP, even in the case of clean channels? To find the answer, we made additional measurements of *Simultaneous* TCP using WhiteHaul and summarise the result in Table 5.1. From the result, there is a high suspicious internal self-interference between the two links. Another experiment is carried out to confirm the suspicion, as shown in Figure 5.1. We use the (80-80) configuration for the radio card. First, we run *iPerf* on WH-1 and leave it to stabilise. WH-1 is considered an always-on interface. Then we switched WH-2 between idle and active by running *iPerf* at 30 seconds and then waited about 35 seconds. We repeated *iPerf* on WH-2 again at 110 seconds and continue for another 35 seconds.

From Figure 5.1, we found that WH-1 throughput decreased when we run *iPerf* on WH-2 even though WH-1 and WH-2 operate on channels that do not overlap and separate 450 MHz apart. An interesting observation is that there is almost the same number of packets transmitted when both links are active. Originally, we expected that both links could transmit packets concurrently without interfering with each other because they use non-overlapping channels. The experiment shows that carrier sense is being used instead for both radio cards. Related work [11] shows that multiradio

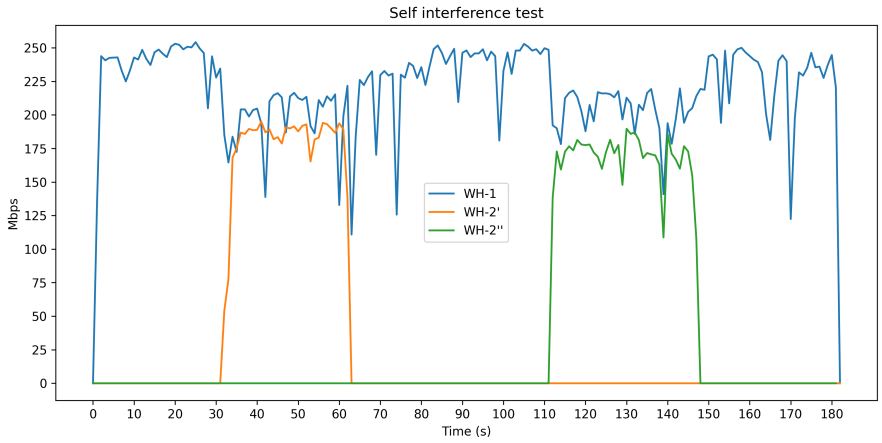


Figure 5.1: Self-Interference for Nonoverlapping Channels

co-existence interference is bound to occur if two antennas are placed close to each other. This founding leads us to derive a mathematical relationship between *Individual* TCP, *Simultaneous* TCP, and MPTCP in the following subsection.

Relationship between *Individual* TCP, *Simultaneous* TCP and MPTCP

It is obvious that throughput obeys the following mathematical relationship.

$$\begin{array}{ccc} \text{Clean MPTCP} & \leqslant & \text{Clean Individual} \\ \rotatebox{90}{$\leqslant$} & & \rotatebox{90}{$\leqslant$} \\ \text{Interference MPTCP} & \leqslant & \text{Interference Individual} \end{array}$$

We can regard

$$\text{Interference MPTCP} \leqslant \min\left(\text{Interference Individual}, \text{Clean MPTCP}\right) \tag{5.1}$$

as a **hard** upper bound of throughput for MPTCP under interference.

Due to possible mutual interference between subflows, we take into account *Simultaneous* TCP, which obeys the following relationship.

$$\begin{array}{ccc} \text{Clean Simultaneous} & \leqslant & \text{Clean Individual} \\ \rotatebox{90}{$\leqslant$} & & \rotatebox{90}{$\leqslant$} \\ \text{Interference Simultaneous} & \leqslant & \text{Interference Individual} \end{array}$$

It turns out that with the same congestion control algorithm used, most of the time MPTCP and *Simultaneous* TCP have the following relationship.

$$\begin{array}{ccc} \text{Clean MPTCP} & \leqslant & \text{Clean Simultaneous} \\ \rotatebox{90}{$\leqslant$} & & \rotatebox{90}{$\leqslant$} \\ \text{Interference MPTCP} & \leqslant & \text{Interference Simultaneous} \end{array}$$

We can regard

$$\text{Interference MPTCP} \leqslant \min\left(\text{Interference Simultaneous}, \text{Clean MPTCP}\right) \tag{5.2}$$

as a **soft** upper bound of throughput for MPTCP under interference.

Hence, we obtain the following relationship.

$$\begin{array}{ccccc} \text{Clean MPTCP} & \leqslant & \text{Clean Simultaneous} & \leqslant & \text{Clean Individual} \\ \curlyvee & & \curlyvee & & \curlyvee \\ \text{Interference MPTCP} & \leqslant & \text{Interference Simultaneous} & \leqslant & \text{Interference Individual} \end{array} \tag{5.3}$$

Note that the above inequalities do not hold for instantaneous throughput, as there may be jitters and delays that affect throughput. However, the inequality holds for average throughput for a sufficiently large period by the law of large numbers.

Channel Quality	Type	WH-1 (Mbps)	WH-2 (Mbps)
Clean	Individual	253	195
Clean	Simultaneous	204	195
Clean	MPTCP	198	193
Interference	Individual	238	139
Interference	Simultaneous	199	138
Interference	MPTCP	195	139

Table 5.2: (80-80) MPTCP CUBIC minRTT average throughput

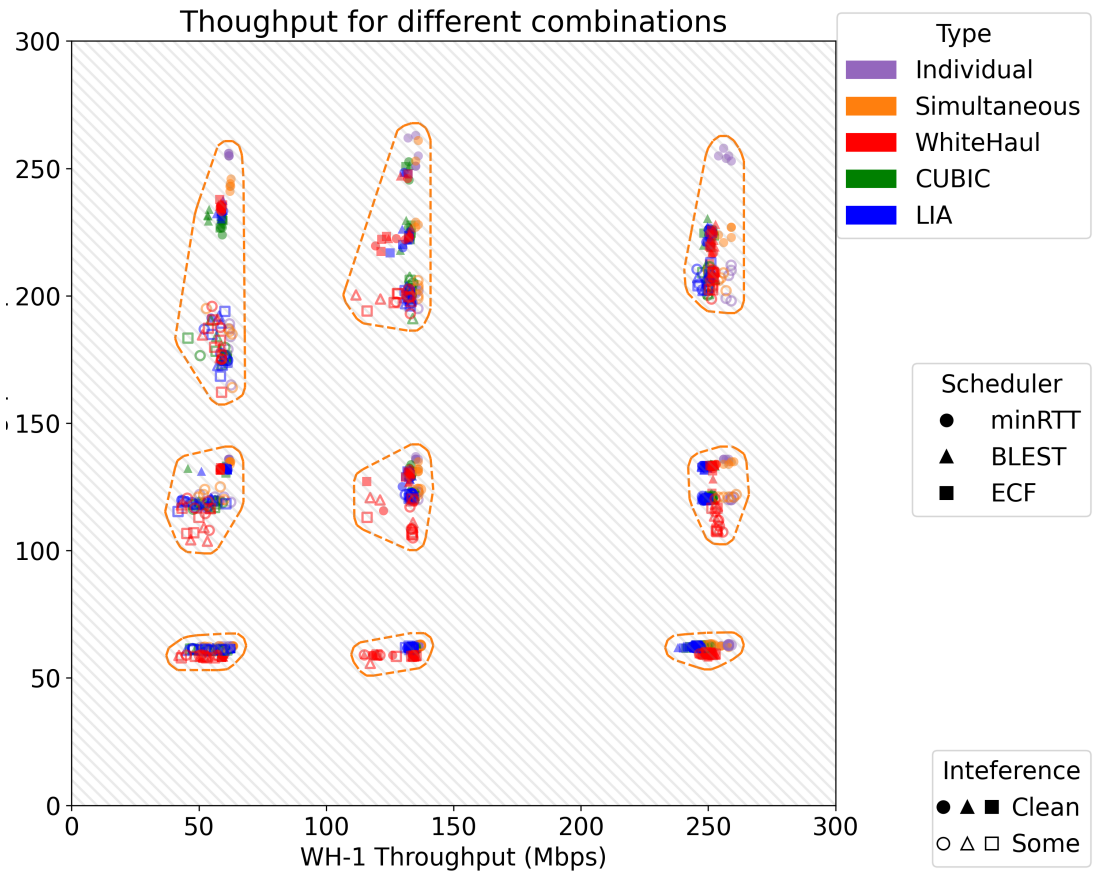


Figure 5.2: Overview of Different Congestion Control and Scheduler Combinations

To confirm our mathematical model, we repeat the experiment with CUBIC as the congestion control algorithm for MPTCP, as it is the congestion control used in both *Individual* TCP and *Simultaneous* TCP. The result in Table 5.2 shows that the throughput of MPTCP with or without interference is approximately 98% of the throughput of *Simultaneous* TCP using the same congestion control. We attribute the 2% difference to the overhead of MPTCP in managing the subflows. The schedulers are responsible for closing the gap between the throughput of MPTCP and the *Simultaneous* TCP, which means that the good throughput, the amount of useful information sent, is close to the maximum throughput available. Figures 5.2 and 5.3 show the throughput of different congestion control and scheduler used for different channel width configurations. Most of the points satisfy the mathematical relationship in (5.3), asserting the correctness and reasonability of the derivation. In summary, we believe that MPTCP congestion control algorithms govern the upper bound of the throughput, and MPTCP schedulers determine the goodput.

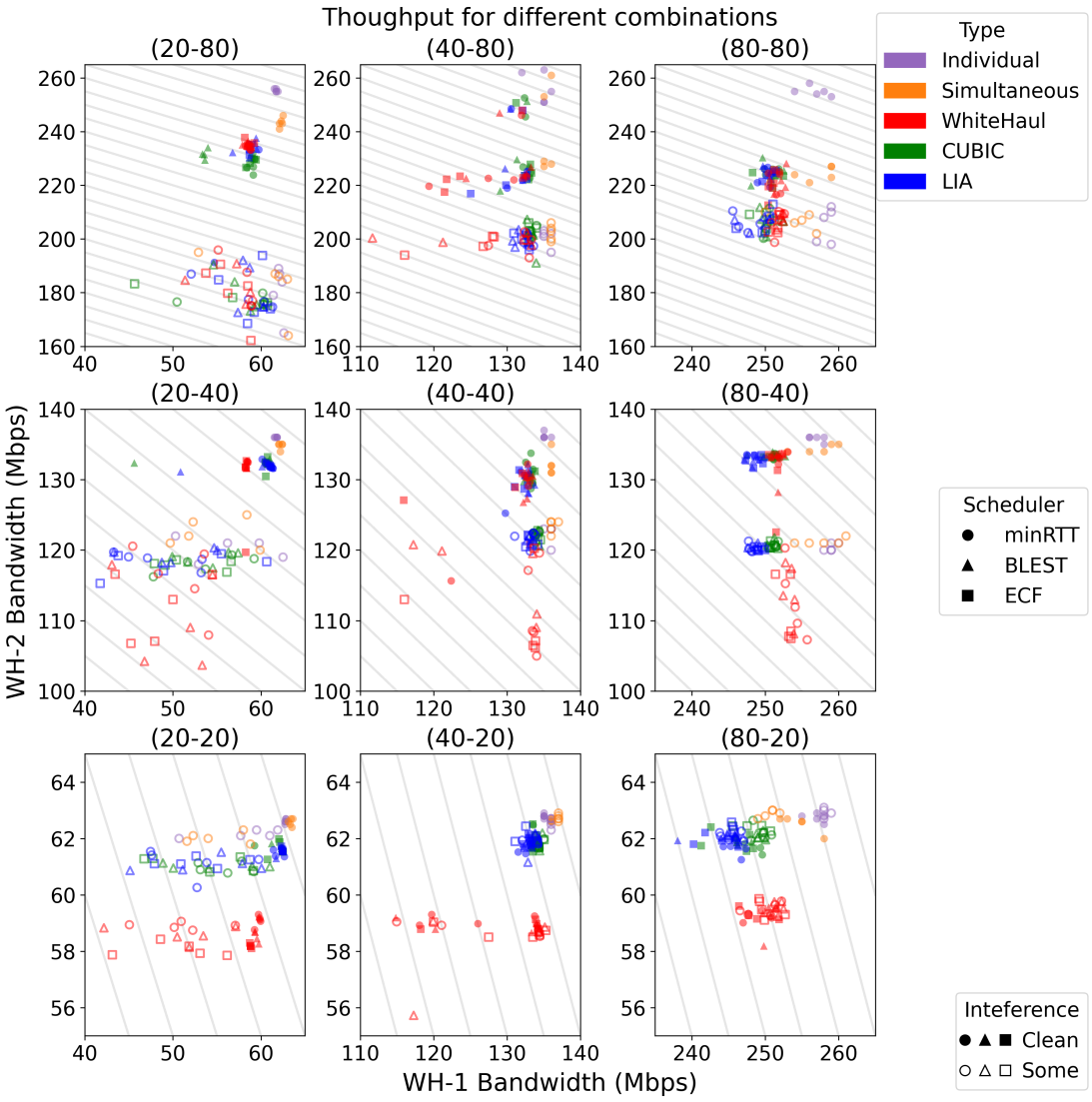


Figure 5.3: An detailed graph of different congestion control and scheduler combinations for each channel width pair

5.2 Raise the Upper Bound

By equation (5.3) if we want to improve the throughput of MPTCP, we need to improve the throughput of *Simultaneous* TCP.

Increase separation of the antennas

The simplest solution is to reduce the mutual interference between the subflows. One method of doing so is to place the two wireless links further apart as the signal strength decreases along the distance. However, it is not always feasible in the case of WhiteHaul. The original WhiteHaul uses a power splitter that combines multiple frequencies together and communicates with the other side using a single antenna. Mutual interference can still occur in power splitting when two frequencies are combined into one. It is one of the hardware flaws that we discovered previously when doing the experiment with a power splitter and a single antenna for transmitting in the Wi-Fi band. The hardware flaw is much more serious as it turns out that when both links are active, only one link can transmit successfully, but with reduced throughput, resulting in a negative aggregation benefit. As we focus on the software aspect of WhiteHaul, we decided to use separate antennas for different frequencies, and the mutual interference between Wi-Fi interfaces is just a weak version of the mutual interference in the hardware. Moreover, as our project's objective is a user-friendly and consumer-level Wi-Fi aggregation system, we prefer a solution that can be easily set up and most wireless devices come with collocated antennas. A solution that requires a large space or dedicated hardware goes against our aim.

Reduce transmission power

Another method of reducing self-interference between two antennas is to reduce transmission power. As the channel quality is determined by SINR, reducing the sender's transmission power will reduce the interference power on the receiver side. Therefore, fewer Wi-Fi frames are lost and reduce the need for retransmission, successively reducing the latency experienced by TCP. However, reducing the transmission power will decrease the incoming signal power at the receiver of the current channel. Recall the Shannon-Hartley theorem[43, 44], notice that if the transmission power of the sender on both subflows is reduced by the same amount, the resulting SINR is less than the original SINR, resulting in lower throughput. For example, if signal power, interference power, and noise are all 8 dbm, the original SINR is thus $\frac{8}{8+8} = \frac{1}{2}$. If the transmission power is reduced by half, then the new SINR is $\frac{4}{4+8} = \frac{1}{3}$ which is less than the original SINR.

We conducted an experiment with a change in the transmission powers of radio cards. We study two cases: define "High" for all transmission powers of radio cards set at 17 dbm, and "Low" for all transmission powers of radio cards set at 14 dbm. We carried out the experiment using the (80-80) channel width configuration with WhiteHaul for MPTCP. The result in Table 5.3 shows that reducing transmission power is not worth it.

Channel Quality	High	Low
Clean	0.72	0.66
Interference	0.27	0.13

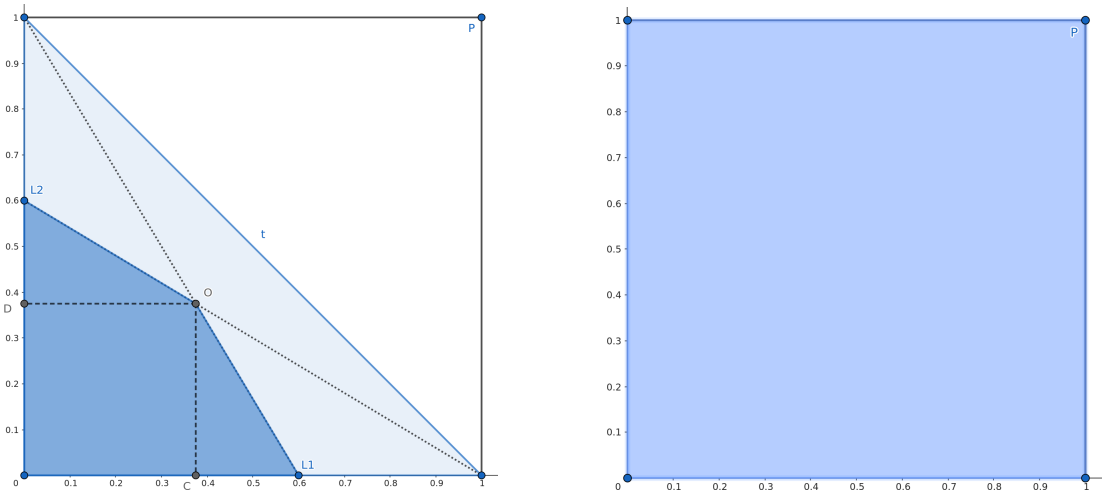
Table 5.3: Aggregation benefit for different transmission power

Better optimising objective

For any multipath aggregation solution, we have the same objective function to maximise: the sum of the throughput of all subflows. In a closed network environment where all links are fully utilised, increasing the throughput of one link will inevitably decrease the throughput of another link. Current Congestion Control algorithms have the assumption that the overall throughput of the closed network environment remains unchanged, i.e. conserved. However, it is not the case when there is mutual interference between subflows where the maximum achievable throughput declines if both links are used. For uncoupled congestion control algorithms, aggressively increasing the CWND for one subflow may drastically decrease the overall maximum achievable throughput.

Let W_i be the send window of the subflow i and T_i be the resulting throughput of the subflow (i). Assume that the interference between links is modelled by CSMA/CA in IEEE 802.11 with exponential back-off, the feasible region is the dark blue area in Figure 5.4a as formulated in [24]. It is shown that there is a bijective function that maps the set of W_i to the set of T_i if it lies in the feasible region. If the set of W_i lies outside the feasible region, then the network response of CSMA/CA will collapse to an equilibrium position on the boundary of the feasible region. Similarly, if there is absolutely no interference between links, such as two independent ethernet cables, then all possible W_i in the square is a feasible configuration, as shown in 5.4b. In fact, the maximum aggregated throughput is at the point P $(1,1)$ which is the sum of the individual throughput of two links. Our situation lies between CSMA/CA and Ethernet. The former acts like co-channel interference between subflows while the latter is isolated interference-free subflows. [24] shows the existence and uniqueness of the optimal point O and provides a maximising sequence to rapidly converge to the point O using a gradient algorithm for Figure 5.4a. The problem is as follows: Can we have an efficient maximising sequence to reach the optimal equilibrium point and remain stable subject to network jitters? This is indeed hard, and more investigation is required to mathematically model the coexistence interference between two Wi-Fi antennas.

The work[24] also shows that the light blue region is achievable by TDMA. The gap between the light grey and dark grey regions is the capacity toll paid due to the adoption of distributed CSMA/CA coordination (dark grey region) instead of a centralised scheduler using TDMA (light grey region). It gives us the idea that we can allocate time for each subflow to send data so that no two subflows send data simultaneously[33]. However, the current 802.11 Wi-Fi standard uses CSMA/CA and requires a change in hardware and protocol design to adopt TDMA. [50] gives the example using alternative wireless medium access (AWMA) and packet traffic arbitration (PTA) to schedule multiple radios in the time domain to ensure that they do not overlap. For now, we treat it as infeasible because it is not readily accessible to customers.



(a) Wi-Fi CSMA (b) Ethernet

Figure 5.4: Feasible region for two subflows with normalised throughput

5.3 Improving the Congestion Control Algorithm

Addressing the Problems in WhiteHaul

We have developed a novel congestion control algorithm named TCP Edinburgh, which is based on WhiteHaul with some adjustment. Firstly, we set the initial SSTHRESH to *TCP_INFINITE_SSTHRESH*, which is consistent with other TCP variants. To avoid overshooting an optimal send rate during slow start, we employ a modified slow start algorithm called HyStart[15], which is already used by the Linux kernel in the CUBIC implementation[26]. However, we use an updated and enhanced version called Hystart++[2] to address network jitter and reduce false positives. Meanwhile, we reimplement the pseudocode in the WhiteHaul paper in a more concise way and allow CWND to increase after experiencing a spike in RTT.

To compare the performance of our TCP Edinburgh and WhiteHaul, we conducted an experiment using (80-80) channel width configuration and calculate the aggregation benefit of both algorithms. In Figure 5.5, Edinburgh has a higher aggregation benefit than WhiteHaul in both the clean channel and interference cases. This means that our algorithm is more efficient in aggregating the available bandwidths than WhiteHaul. Figure 5.6 shows that Edinburgh's ultilisation is closer to the upper bound, *Simultaneous* TCP throughput.

Use alternative congestion control algorithm

WhiteHaul is based on TCP Illinois [29] which is suitable for a long-fat network, but it is not designed for wireless networks. [6] gives a behaviour analysis of TCP different TCP variants and reveals that TCP Illinois gives the worst throughput in the wireless network and the network that changes bandwidth. On the contrary, [6] shows TCP CUBIC performs moderately in different networking scenarios. Considering the nature of Wi-Fi networks, we suggest using a congestion control that can tackle variation in

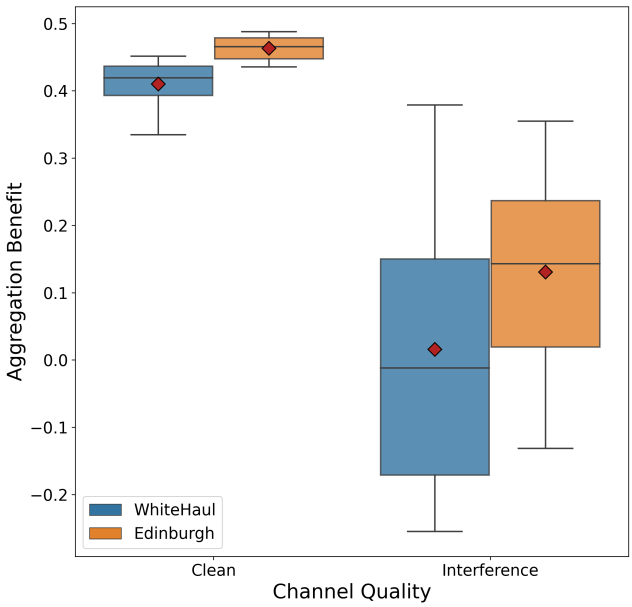


Figure 5.5: (80-80) Aggregation Benefit for WhiteHaul and Edinburgh

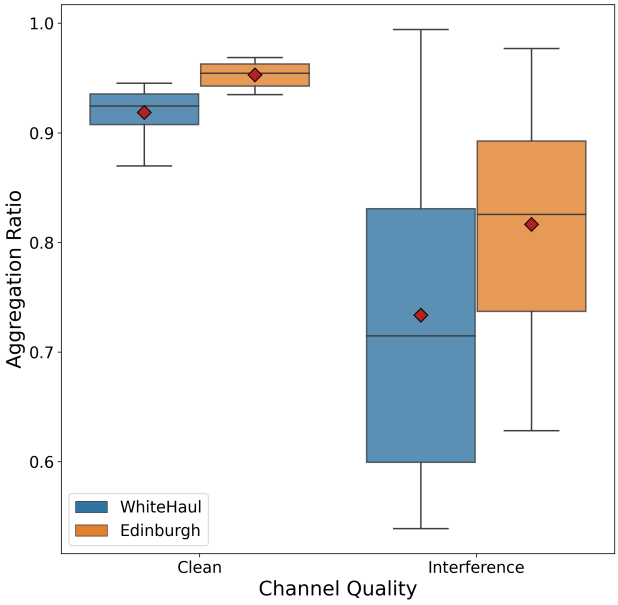


Figure 5.6: (80-80) Aggregation Ratio for WhiteHaul and Edinburgh

network conditions, and CUBIC is a good candidate. It matches the observation in the previous section 4.3 that CUBIC performs better than Illinois in clean channel cases.

Chapter 6

Conclusion

6.1 Summary

In this project, we conducted an in-depth investigation into the performance of WhiteHaul in Wi-Fi networks, particularly in the presence of interference. We showed that changing the MPTCP scheduler does not make a noticeable difference in the aggregation throughput of WhiteHaul. In contrast, we showed that changing the MPTCP congestion control may lead to an improvement in the aggregation throughput of WhiteHaul. Combining the above two findings, we show that MPTCP congestion controls play an important role in governing the maximum total aggregation throughput, while MPTCP schedulers limit the application's goodput. We derived an upper bound for the aggregation throughput of MPTCP in the wireless network.

By looking at the internal workings of WhiteHaul, we discovered the design flaw in WhiteHaul's implementation inside the Linux kernel. We list the problems and provide patches for each issue. In addition, we propose improvements to the WhiteHaul algorithm. Apart from the Whitehaul aspect, we also provided several potential improvements and discussed their limitations and feasibilities.

6.2 Challenges and Lessons Learnt

We encountered a myriad of difficulties in conducting the experiments. The primary issues we faced were related to hardware, which consumed most of our time to debug. Mentioned in 5.2, we originally used a power splitter to combine the signals of each radio card in one and communicate with a single antenna. However, we observed unexpected behaviour, leading to a considerable decrease in throughput. After much effort, we discovered that the problem was due to the power splitter. Other problems, such as loose cables and defective antennas, obstructed our efforts to obtain sensible experimental results.

During the research, we were taught to copy with the uncertainties present in networks, particularly with Wi-Fi, which is highly sensitive to the surrounding physical environment. Any background movement or adjacent channel activity will introduce a spike in

the collected data and may take longer for the network to restore to its original state. To minimise interference from human activities, we had to carry out the experiments during the night and on weekends.

Throughout our work, we continued to expand our understanding of networking. Although we had learnt TCP in undergraduate networking courses, our understanding was rudimentary and we were not familiar with the details of the internal implementation at the operating system level. When working on the MPTCP congestion controls in the Linux kernel and writing eBPF programs to hook up the kernel functions, we learnt a lot about the low-level aspects.

We encountered a major difficulty in the software component of the project, which involved creating TCP Edinburgh. Our slow-start algorithm, HyStart++, is not yet finalised, as it is still in the development stages. Although Cloudflare has a library called *quiche* that supports HyStart++ on the QUIC protocol written in Rust, there are no existing C/C++ implementations online. We have to thoroughly study the specifications and write our own implementation of Hystart++ in the Linux kernel from strech.

6.3 Future Work

More experiments are needed to explore more uncoupled TCP congestion control variants to determine which variant performs modestly under a wide variety of Wi-Fi network conditions. We need a congestion control that is robust to rapid changes in the network situation, as Wi-Fi is highly unstable.

It would be interesting to study how different levels of channel quality affect the aggregation throughput. By gradually altering the signal power or adjusting the interference power, or both, we can study how various levels of interference and transmission power influence total throughput. It will also be helpful to develop a more realistic mathematical model for different types of interference[45], and to take these into account when designing an aggregation system.

So far our paper has used only two Wi-FI interfaces, it would be insightful to verify that three or more interfaces can achieve decent aggregation throughput and do not exhibit chaotic behaviour or performance regression.

While we assume backhaul traffic in our settings, which is highly assymmetric and almost unidirectional, real-life scenarios involve more symmetric traffic. We recommend exploring bidirectional traffic in Wi-Fi settings, as Wi-Fi operates in half-duplex mode, which may result in high collisions in data transmission between the sender and receiver.

Our paper has only explored IEEE 802.11ac Wi-Fi 5 channel aggregation in the 5GHz band. Since 2020, the 6GHz band has been available for IEEE 802.11ax Wi-Fi 6E with higher data transmission rates[3, 28]. There is significant opportunity to exploit channels in both the 5GHz and 6GHz spectrum.

Bibliography

[1] Atef Abdrabou and Monika Prakash. "Experimental Performance Study of Multipath TCP over Heterogeneous Wireless Networks". In: *2016 IEEE 41st Conference on Local Computer Networks (LCN)*. 2016 IEEE 41st Conference on Local Computer Networks (LCN). Nov. 2016, pp. 172–175. DOI: 10.1109/LCN.2016.35.

[2] Praveen Balasubramanian, Yi Huang, and Matt Olson. *HyStart++: Modified Slow Start for TCP*. Internet Draft draft-ietf-tcpm-hystartplusplus-14. Num Pages: 10. Internet Engineering Task Force, Feb. 27, 2023. URL: https://datatracker.ietf.org/doc/draft-ietf-tcpm-hystartplusplus (visited on 04/10/2023).

[3] Dwaipayan Bandyopadhyay et al. "Network Throughput Improvement in Wi-Fi 6 over Wi-Fi 5: A Comparative Performance Analysis". In: *2023 International Conference on Computer, Electrical & Communication Engineering (ICCECE)*. 2023 International Conference on Computer, Electrical & Communication Engineering (ICCECE). ISSN: 2768-0576. Jan. 2023, pp. 1–6. DOI: 10.1109/ICCECE51049.2023.10085684.

[4] Ethan Blanton, Vern Paxson, and Mark Allman. *TCP Congestion Control*. Request for Comments RFC 5681. Num Pages: 18. Internet Engineering Task Force, Sept. 2009. DOI: 10.17487/RFC5681. URL: https://datatracker.ietf.org/doc/rfc5681 (visited on 03/26/2023).

[5] David Borman et al. *TCP Extensions for High Performance*. Request for Comments RFC 7323. Num Pages: 49. Internet Engineering Task Force, Sept. 2014. DOI: 10.17487/RFC7323. URL: https://datatracker.ietf.org/doc/rfc7323 (visited on 04/01/2023).

[6] C. Callegari et al. "Behavior analysis of TCP Linux variants". In: *Computer Networks* 56.1 (Jan. 12, 2012), pp. 462–476. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2011.10.002. URL: https://www.sciencedirect.com/science/article/pii/S1389128611003677 (visited on 04/10/2023).

[7] Yung-Chih Chen et al. "A measurement-based study of MultiPath TCP performance over wireless networks". In: *Proceedings of the 2013 conference on Internet measurement conference*. IMC '13. New York, NY, USA: Association for Computing Machinery, Oct. 23, 2013, pp. 455–468. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504751. URL: https://dl.acm.org/doi/10.1145/2504730.2504751 (visited on 03/26/2023).

[8] Shuo Deng et al. "WiFi, LTE, or Both? Measuring Multi-Homed Wireless Internet Performance". In: *Proceedings of the 2014 Conference on Internet Mea-*

surement Conference. IMC '14. New York, NY, USA: Association for Computing Machinery, Nov. 5, 2014, pp. 181–194. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663727. URL: https://dl.acm.org/doi/10.1145/2663716.2663727 (visited on 04/11/2023).

[9] Jingpu Duan, Zhi Wang, and Chuan Wu. "Responsive multipath TCP in SDN-based datacenters". In: *2015 IEEE International Conference on Communications (ICC)*. 2015 IEEE International Conference on Communications (ICC). ISSN: 1938-1883. June 2015, pp. 5296–5301. DOI: 10.1109/ICC.2015.7249165.

[10] Wesley Eddy. *Transmission Control Protocol (TCP)*. Request for Comments RFC 9293. Num Pages: 98. Internet Engineering Task Force, Aug. 2022. DOI: 10.17487/RFC9293. URL: https://datatracker.ietf.org/doc/rfc9293 (visited on 03/23/2023).

[11] Arsham Farshad, Mahesh K. Marina, and Francisco Garcia. "Experimental investigation of coexistence interference on multi-radio 802.11 platforms". In: *2012 10th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt)*. 2012 10th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt). May 2012, pp. 293–298.

[12] Simone Ferlin, Thomas Dreibholz, and Özgü Alay. "Multi-path transport over heterogeneous wireless networks: Does it really pay off?" In: *2014 IEEE Global Communications Conference*. 2014 IEEE Global Communications Conference. ISSN: 1930-529X. Dec. 2014, pp. 4807–4813. DOI: 10.1109/GLOCOM.2014.7037567.

[13] Simone Ferlin et al. "BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks". In: *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. 2016 IFIP Networking Conference (IFIP Networking) and Workshops. May 2016, pp. 431–439. DOI: 10.1109/IFIPNetworking.2016.7497206.

[14] Alan Ford et al. *TCP Extensions for Multipath Operation with Multiple Addresses*. Request for Comments RFC 8684. Num Pages: 68. Internet Engineering Task Force, Mar. 2020. DOI: 10.17487/RFC8684. URL: https://datatracker.ietf.org/doc/rfc8684 (visited on 03/23/2023).

[15] Sangtae Ha and Injong Rhee. "Taming the elephants: New TCP slow start". In: *Computer Networks* 55.9 (June 23, 2011), pp. 2092–2110. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2011.01.014. URL: https://www.sciencedirect.com/science/article/pii/S1389128611000363 (visited on 04/10/2023).

[16] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *ACM SIGOPS Operating Systems Review* 42.5 (July 1, 2008), pp. 64–74. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. URL: https://dl.acm.org/doi/10.1145/1400097.1400105 (visited on 03/26/2023).

[17] *Home — Whitehaul*. URL: https://www.whitehaul.com/ (visited on 03/20/2023).

[18] "IEEE Standard for Information technology– Telecommunications and information exchange between systemsLocal and metropolitan area networks– Specific requirements–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications–Amendment 4: Enhancements for Very High

Throughput for Operation in Bands below 6 GHz." In: *IEEE Std 802.11ac-2013 (Amendment to IEEE Std 802.11-2012, as amended by IEEE Std 802.11ae-2012, IEEE Std 802.11aa-2012, and IEEE Std 802.11ad-2012)* (Dec. 2013). Conference Name: IEEE Std 802.11ac-2013 (Amendment to IEEE Std 802.11-2012, as amended by IEEE Std 802.11ae-2012, IEEE Std 802.11aa-2012, and IEEE Std 802.11ad-2012), pp. 1–425. DOI: 10.1109/IEEESTD.2013.6687187.

[19] "IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks–Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications". In: *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)* (Feb. 2021). Conference Name: IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016), pp. 1–4379. DOI: 10.1109/IEEESTD.2021.9363693.

[20] "IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks–Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 1: Enhancements for High-Efficiency WLAN". In: *IEEE Std 802.11ax-2021 (Amendment to IEEE Std 802.11-2020)* (May 2021). Conference Name: IEEE Std 802.11ax-2021 (Amendment to IEEE Std 802.11-2020), pp. 1–767. DOI: 10.1109/IEEESTD.2021.9442429.

[21] Jana Iyengar et al. *Architectural Guidelines for Multipath TCP Development*. Request for Comments RFC 6182. Num Pages: 28. Internet Engineering Task Force, Mar. 2011. DOI: 10.17487/RFC6182. URL: https://datatracker.ietf.org/doc/rfc6182 (visited on 04/11/2023).

[22] Dominik Kaspar. "Multipath Aggregation of Heterogeneous Access Networks". Accepted: 2013-03-12T08:05:47Z. Doctoral thesis. 2011. URL: https://www.duo.uio.no/handle/10852/9034 (visited on 03/25/2023).

[23] Mohamed M. Kassem et al. "WhiteHaul: an efficient spectrum aggregation system for low-cost and high capacity backhaul over white spaces". In: *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*. MobiSys '20. New York, NY, USA: Association for Computing Machinery, June 15, 2020, pp. 338–351. ISBN: 978-1-4503-7954-0. DOI: 10.1145/3386901.3388950. URL: https://dl.acm.org/doi/10.1145/3386901.3388950 (visited on 03/20/2023).

[24] Rafael Laufer and Leonard Kleinrock. "The Capacity of Wireless CSMA/CA Networks". In: *IEEE/ACM Transactions on Networking* 24.3 (June 2016). Conference Name: IEEE/ACM Transactions on Networking, pp. 1518–1532. ISSN: 1558-2566. DOI: 10.1109/TNET.2015.2415465.

[25] Yeon-sup Lim et al. "ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths". In: *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. CoNEXT '17. New York, NY, USA: Association for Computing Machinery, Nov. 28, 2017, pp. 147–159. ISBN: 978-1-4503-5422-6. DOI: 10.1145/3143361.3143376. URL: https://dl.acm.org/doi/10.1145/3143361.3143376 (visited on 04/08/2023).

[26] Torvalds Linus. *Linux Kernel: [TCP] CUBIC v2.3*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ae27e98a51526595837ab7498b23d6478a198960 (visited on 04/12/2023).

[27] Torvalds Linus. *Linux Kernel: [TCP]: make cubic the default*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=597811ec167fa01c926a0957a91d9e39baa30e64 (visited on 04/09/2023).

[28] Ruofeng Liu and Nakjung Choi. "A First Look at Wi-Fi 6 in Action: Throughput, Latency, Energy Efficiency, and Security". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7.1 (Mar. 2, 2023), 25:1–25:25. DOI: 10.1145/3579451. URL: https://dl.acm.org/doi/10.1145/3579451 (visited on 04/12/2023).

[29] Shao Liu, Tamer Başar, and R. Srikant. "TCP-Illinois: a loss and delay-based congestion control algorithm for high-speed networks". In: *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*. valuetools '06. New York, NY, USA: Association for Computing Machinery, Oct. 11, 2006, 55–es. ISBN: 978-1-59593-504-5. DOI: 10.1145/1190095.1190166. URL: https://dl.acm.org/doi/10.1145/1190095.1190166 (visited on 04/06/2023).

[30] *LoRa PHY — Semtech*. URL: https://www.semtech.com/lora/what-is-lora (visited on 04/12/2023).

[31] Rajesh Mahindra et al. "A practical traffic management system for integrated LTE-WiFi networks". In: *Proceedings of the 20th annual international conference on Mobile computing and networking*. MobiCom '14. New York, NY, USA: Association for Computing Machinery, Sept. 7, 2014, pp. 189–200. ISBN: 978-1-4503-2783-1. DOI: 10.1145/2639108.2639120. URL: https://dl.acm.org/doi/10.1145/2639108.2639120 (visited on 04/11/2023).

[32] Imtiaz Mahmud, Tabassum Lubna, and You-Ze Cho. "Performance Evaluation of MPTCP on Simultaneous Use of 5G and 4G Networks". In: *Sensors* 22.19 (Jan. 2022). Number: 19 Publisher: Multidisciplinary Digital Publishing Institute, p. 7509. ISSN: 1424-8220. DOI: 10.3390/s22197509. URL: https://www.mdpi.com/1424-8220/22/19/7509 (visited on 03/28/2023).

[33] Guowang Miao et al. *Fundamentals of Mobile Data Networks*. Cambridge: Cambridge University Press, 2016. ISBN: 978-1-107-14321-0. DOI: 10.1017/CBO9781316534298. URL: https://www.cambridge.org/core/books/fundamentals-of-mobile-data-networks/D46688899BDC64F4421967A916548433 (visited on 04/12/2023).

[34] Christoph Paasch et al. "Experimental evaluation of multipath TCP schedulers". In: *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*. CSWS '14. New York, NY, USA: Association for Computing Machinery, Aug. 18, 2014, pp. 27–32. ISBN: 978-1-4503-2991-0. DOI: 10.1145/2630088.2631977. URL: https://dl.acm.org/doi/10.1145/2630088.2631977 (visited on 03/26/2023).

[35] Christoph Paasch et al. "Exploring mobile/WiFi handover with multipath TCP". In: *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*. CellNet '12. New York, NY, USA:

Association for Computing Machinery, Aug. 13, 2012, pp. 31–36. ISBN: 978-1-4503-1475-6. DOI: 10.1145/2342468.2342476. URL: https://dl.acm.org/doi/10.1145/2342468.2342476 (visited on 04/11/2023).

[36] Mijanur Rahaman Palash and Kang Chen. "MPWiFi: Synergizing MPTCP Based Simultaneous Multipath Access and WiFi Network Performance". In: *IEEE Transactions on Mobile Computing* 19.1 (Jan. 2020). Conference Name: IEEE Transactions on Mobile Computing, pp. 142–158. ISSN: 1558-0660. DOI: 10.1109/TMC.2018.2889059.

[37] Matthew Podolsky et al. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. Request for Comments RFC 2883. Num Pages: 17. Internet Engineering Task Force, July 2000. DOI: 10.17487/RFC2883. URL: https://datatracker.ietf.org/doc/rfc2883 (visited on 04/01/2023).

[38] Costin Raiciu, Mark J. Handley, and Damon Wischik. *Coupled Congestion Control for Multipath Transport Protocols*. Request for Comments RFC 6356. Num Pages: 12. Internet Engineering Task Force, Oct. 2011. DOI: 10.17487/RFC6356. URL: https://datatracker.ietf.org/doc/rfc6356 (visited on 04/07/2023).

[39] Costin Raiciu 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). 2012, pp. 399–412. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/raiciu (visited on 04/01/2023).

[40] Costin Raiciu et al. "Data center networking with multipath TCP". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. New York, NY, USA: Association for Computing Machinery, Oct. 20, 2010, pp. 1–6. ISBN: 978-1-4503-0409-2. DOI: 10.1145/1868447.1868457. URL: https://dl.acm.org/doi/10.1145/1868447.1868457 (visited on 04/11/2023).

[41] Costin Raiciu et al. "Improving datacenter performance and robustness with multipath TCP". In: *Proceedings of the ACM SIGCOMM 2011 conference*. SIGCOMM '11. New York, NY, USA: Association for Computing Machinery, Aug. 15, 2011, pp. 266–277. ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018467. URL: https://dl.acm.org/doi/10.1145/2018436.2018467 (visited on 04/11/2023).

[42] Pasi Sarolahti and Alexey Kuznetsov. "Congestion Control in Linux {TCP}". In: 2002 USENIX Annual Technical Conference (USENIX ATC 02). 2002. URL: https://www.usenix.org/conference/2002-usenix-annual-technical-conference/congestion-control-linux-tcp (visited on 03/28/2023).

[43] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (July 1948). Conference Name: The Bell System Technical Journal, pp. 379–423. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[44] C.E. Shannon. "Communication in the Presence of Noise". In: *Proceedings of the IRE* 37.1 (Jan. 1949). Conference Name: Proceedings of the IRE, pp. 10–21. ISSN: 2162-6634. DOI: 10.1109/JRPROC.1949.232969.

[45] Gaotao Shi and Keqiu Li. “Interference Model and Measurement”. In: *Signal Interference in WiFi and ZigBee Networks*. Ed. by Gaotao Shi and Keqiu Li. Wireless Networks. Cham: Springer International Publishing, 2017, pp. 29–44. ISBN: 978-3-319-47806-7. DOI: 10.1007/978-3-319-47806-7_3. URL: https://doi.org/10.1007/978-3-319-47806-7_3 (visited on 04/12/2023).

[46] Tanya Shreedhar et al. “QAware: A Cross-Layer Approach to MPTCP Scheduling”. In: *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. 2018 IFIP Networking Conference (IFIP Networking) and Workshops. May 2018, pp. 1–9. DOI: 10.23919/IFIPNetworking.2018.8696843.

[47] Zhenyu Song, Longfei Shangguan, and Kyle Jamieson. “Wi-Fi Goes to Town: Rapid Picocell Switching for Wireless Transit Networks”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. New York, NY, USA: Association for Computing Machinery, Aug. 7, 2017, pp. 322–334. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098846. URL: https://dl.acm.org/doi/10.1145/3098822.3098846 (visited on 04/11/2023).

[48] *Wireless Interference and Multipath TCP*. URL: https://jon.tsp.io/mscncs/ (visited on 03/20/2023).

[49] Damon Wischik et al. “Design, Implementation and Evaluation of Congestion Control for Multipath {TCP}”. In: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11). 2011. URL: https://www.usenix.org/conference/nsdi11/design-implementation-and-evaluation-congestion-control-multipath-tcp (visited on 04/01/2023).

[50] Jing Zhu et al. “Multi-Radio Coexistence: Challenges and Opportunities”. In: *2007 16th International Conference on Computer Communications and Networks*. 2007 16th International Conference on Computer Communications and Networks. ISSN: 1095-2055. Aug. 2007, pp. 358–364. DOI: 10.1109/ICCCN.2007.4317845.

Appendix A

WhiteHaul Source Code

```
1 /*
2  * TCP Illinois congestion control.
3  * Home page:
4  *
       http://www.ews.uiuc.edu/~shaoliu/tcpillinois/index.html
5  *
6  * The algorithm is described in:
7  * "TCP-Illinois: A Loss and Delay-Based Congestion
       Control Algorithm
8  *  for High-Speed Networks"
9  *
       http://tamerbasar.csl.illinois.edu/LiuBasarSrikantPerfEvalArtJun
10 *
11 * Implemented from description in paper and ns-2
       simulation.
12 * Copyright (C) 2007 Stephen Hemminger
       <shemminger@linux-foundation.org>
13 */
14
15 #include <linux/module.h>
16 #include <linux/skbuff.h>
17 #include <linux/inet_diag.h>
18 #include <asm/div64.h>
19 #include <net/tcp.h>
20 #include <linux/ktime.h>
21 #include <linux/time.h>
22
23 //#define ADVERTISED_CWND 10    // MKS: need to be
       replace with a callback
24 #define RTT_BASE_CLAMP  100 // RTT can't be less than
       this value (us)
25 #define ALPHA_SHIFT 7
```

```
26 #define ALPHA_SCALE (1u<<ALPHA_SHIFT)
27 #define ALPHA_MIN ((3*ALPHA_SCALE)/10)  /* ~0.3 */
28 #define ALPHA_MAX (50*ALPHA_SCALE)  /* 10.0 */
29 #define ALPHA_BASE  ALPHA_SCALE   /* 1.0 */
30 #define RTT_MAX   (U32_MAX / ALPHA_MAX) /* 3.3 secs */
31
32 // We are not using any of these at the moments
33 #define BETA_SHIFT  6
34 #define BETA_SCALE  (1u<<BETA_SHIFT)
35 #define BETA_MIN  (BETA_SCALE/10)   /* 0.125 */
36 #define BETA_MAX  (BETA_SCALE/10)   /* 0.5 */
37 #define BETA_BASE BETA_MAX
38
39 //static int win_thresh __read_mostly = 15;
40 static int win_thresh __read_mostly = 10; /* MKS */
41 module_param(win_thresh, int, 0);
42 MODULE_PARM_DESC(win_thresh, "Window threshold for
      starting adaptive sizing");
43
44 //static int theta __read_mostly = 5;
45 static int theta __read_mostly = 2; /* MKS */
46 module_param(theta, int, 0);
47 MODULE_PARM_DESC(theta, "# of fast RTT's before full
      growth");
48
49 static int ADVERTISED_CWND_1 = 100; /* These are the two
      advertised windows */
50 module_param(ADVERTISED_CWND_1, int, 0644);
51 MODULE_PARM_DESC(ADVERTISED_CWND_1, "Advertised window
      for subflow #1");
52
53 static int ADVERTISED_CWND_2 = 100; /* These are the two
      advertised windows */
54 module_param(ADVERTISED_CWND_2, int, 0644);
55 MODULE_PARM_DESC(ADVERTISED_CWND_2, "Advertised window
      for subflow #2");
56
57 static int ADVERTISED_CWND_3 = 100; /* These are the two
      advertised windows */
58 module_param(ADVERTISED_CWND_3, int, 0644);
59 MODULE_PARM_DESC(ADVERTISED_CWND_3, "Advertised window
      for subflow #3");
60
61
62 /* TCP Illinois Parameters */
63 struct illinois {
```

```
64   u64 sum_rtt;  /* sum of rtt's measured within last rtt
       */
65   u16 cnt_rtt;  /* # of rtts measured within last rtt */
66   u32 base_rtt; /* min of all rtt in usec */
67   u32 max_rtt;  /* max of all rtt in usec */
68   u32 end_seq;  /* right edge of current RTT */
69   u32 alpha;    /* Additive increase */
70   u32 beta;   /* Muliplicative decrease */
71   u16 acked;    /* # packets acked by current ACK */
72   u8  rtt_above;  /* average rtt has gone above
       threshold */
73   u8  rtt_low;  /* # of rtts measurements below
       threshold */
74   u8    cnt_round;      /* # of round that whole flight
       packets get acked */
75   u8    curr_state;     /* curret tcp congestion control
       state */
76   //  u32 avg_delay;  /* Average queuing delay */
77   //  u32 max_delay;  /* Maximum queuing delay */
78 };
79
80 // static void rtt_base_reset(struct sock *sk, u32 da) {
81 //  struct illinois *ca = inet_csk_ca(sk);
82 //          ca->base_rtt = 0x7fffffff;  /* MKS: we can
     use da instead */
83 //          ca->cnt_round = 0;
84 // }
85
86 static void rtt_reset(struct sock *sk)
87 {
88   struct tcp_sock *tp = tcp_sk(sk);
89   struct illinois *ca = inet_csk_ca(sk);
90
91   ca->end_seq = tp->snd_nxt;
92   ca->cnt_rtt = 0;
93   ca->sum_rtt = 0;
94
95   /* TODO: age max_rtt? */
96 }
97
98 static void tcp_illinois_init(struct sock *sk)
99 {
100  struct illinois *ca = inet_csk_ca(sk);
101
102  ca->alpha = ALPHA_MAX;
103  ca->beta = BETA_BASE;
```

```
104   ca->base_rtt = 0x7fffffff;
105   ca->max_rtt = 0;
106
107   ca->acked = 0;
108   ca->rtt_low = 0;
109   ca->rtt_above = 0;
110         ca->curr_state = TCP_CA_Open; // MKS
111         tcp_sk(sk)->snd_ssthresh = 20;
112   rtt_reset(sk);
113 }
114
115 /* Measure RTT for each ack. */
116 static void tcp_illinois_acked(struct sock *sk, const
      struct ack_sample *sample)
117 {
118   struct illinois *ca = inet_csk_ca(sk);
119   s32 rtt_us = sample->rtt_us;
120   struct timeval ts;
121         do_gettimeofday(&ts);
122         s64 timestamp = (ts.tv_sec) * 1000000 +
            (ts.tv_usec);
123
124   //printk(KERN_INFO "%s: %pI4:%d -> %pI4:%d, timestamp
        %lld and rtt %u \n", __func__,&((struct inet_sock
        *) sk)->inet_saddr, ntohs(((struct inet_sock *)
        sk)->inet_sport), &((struct inet_sock *)
        sk)->inet_daddr, ntohs(((struct inet_sock *)
        sk)->inet_dport), timestamp, rtt_us);
125         //printk(KERN_INFO "At time %lld: RTT = %u\n",
            timestamp, rtt_us);
126
127   ca->acked = sample->pkts_acked;
128
129   /* dup ack, no rtt sample */
130   if (rtt_us < 0)
131     return;
132
133   /* ignore bogus values, this prevents wraparound in
        alpha math */
134   if (rtt_us > RTT_MAX)
135     rtt_us = RTT_MAX;
136
137   /* keep track of minimum RTT seen so far */
138   if (ca->base_rtt > rtt_us)
139         { // MKS: rtt_us should be bigger than
            RTT_BASE_CLAMP
```

```
140             if (rtt_us > RTT_BASE_CLAMP)
141               ca->base_rtt = rtt_us;
142           }
143
144         /* and max */
145         if (ca->max_rtt < rtt_us)
146           ca->max_rtt = rtt_us;
147
148   ++ca->cnt_rtt;
149   ca->sum_rtt += rtt_us;
150 }
151
152 /* Maximum queuing delay */
153 static inline u32 max_delay(const struct illinois *ca)
154 {
155   return ca->max_rtt - ca->base_rtt;
156 }
157
158 /* Average queuing delay */
159 static inline u32 avg_delay(const struct illinois *ca)
160 {
161   u64 t = ca->sum_rtt;
162
163   do_div(t, ca->cnt_rtt);
164   return t - ca->base_rtt;
165 }
166
167 /*
168  * Compute value of alpha used for additive increase.
169  * If small window then use 1.0, equivalent to Reno.
170  *
171  * For larger windows, adjust based on average delay.
172  * A. If average delay is at minimum (we are
       uncongested),
173  *    then use large alpha (10.0) to increase faster.
174  * B. If average delay is at maximum (getting congested)
175  *    then use small alpha (0.3)
176  *
177  * The result is a convex window growth curve.
178  */
179 // static u32 alpha(struct illinois *ca, u32 da, u32 dm)
180 // {
181 //  u32 d1 = dm / 100;  /* Low threshold */
182
183 //          if (da <= d1) {
184 //    /* If never got out of low delay zone, then use
```

```
    max */
185 //     if (!ca->rtt_above)
186 //       return ALPHA_MAX;
187 //
188 //     /* Wait for 5 good RTT's before allowing alpha to
    go alpha max.
189 //      * This prevents one good RTT from causing sudden
    window increase.
190 //      */
191 //     if (++ca->rtt_low < theta)
192 //       return ca->alpha;
193 //
194 //     ca->rtt_low = 0;
195 //     ca->rtt_above = 0;
196 //     return ALPHA_MAX;
197 //   }
198 //
199 //   ca->rtt_above = 1;
200 //
201 //   /*
202 //    * Based on:
203 //    *
204 //    *       (dm - d1) amin amax
205 //    * k1 = -------------------
206 //    *            amax - amin
207 //    *
208 //    *         (dm - d1) amin
209 //    * k2 = ----------------  - d1
210 //    *           amax - amin
211 //    *
212 //    *                 k1
213 //    * alpha = ----------
214 //    *             k2 + da
215 //    */
216 //
217 //   dm -= d1;
218 //   da -= d1;
219 //   return (dm * ALPHA_MAX) /
220 //     (dm + (da  * (ALPHA_MAX - ALPHA_MIN)) / ALPHA_MIN);
221 // }
222 //
223 /* We just added another paramters to this function to
    differentiate between two sub-flows */
224 //
225 static u32 alpha(struct illinois *ca, u32 da, u32 dm,
    struct sock *sk, u32 adv_window)
```

```
226 {
227   struct tcp_sock *tp = tcp_sk(sk);
228         u32 qDelayBudget_us = 50000; // usec
229
230         /* queueing delay average (da) is smaller than
               our queueing delay budget */
231         if (da <= qDelayBudget_us) {
232           if (tp->snd_cwnd >= adv_window) {
233             // cwnd is bigger than advertised cwnd,
                   increase slowly
234             return ALPHA_BASE;
235           }
236           else {
237             /* If never got out of low delay zone, then
                   use max */
238             if (!ca->rtt_above)
239               return ALPHA_MAX;
240
241             /* Wait for 5 good RTT's before allowing
                   alpha to go alpha max.
242              * This prevents one good RTT from causing
                    sudden window increase.
243              * Maybe we should remove this part, so we
                    can increase immidiately
244              * need further exploration
245              */
246             if (++ca->rtt_low < theta)
247               return ca->alpha;
248
249             ca->rtt_low = 0;
250             ca->rtt_above = 0;
251             return ALPHA_MAX;
252           }
253         }
254         else {
255           /* mean delay bigger than our delay budget */
256           ca->rtt_above = 1; // move to high rtt zone
257           return ALPHA_BASE;  // send aggresive as we
                 are below target rate
258         }
259 }
260
261 /*
262  * Beta used for multiplicative decrease.
263  * For small window sizes returns same value as Reno
       (0.5)
```

```
264  *
265  * If delay is small (10% of max) then beta = 1/8
266  * If delay is up to 80% of max then beta = 1/2
267  * In between is a linear function
268  */
269 static u32 beta(u32 da, u32 dm)
270 {
271  u32 d2, d3;
272
273  d2 = dm / 10;
274  if (da <= d2)
275    return BETA_MIN;
276
277  d3 = (8 * dm) / 10;
278  if (da >= d3 || d3 <= d2)
279    return BETA_MAX;
280
281  /*
282   * Based on:
283   *
284   *        bmin d3 - bmax d2
285   * k3 = -------------------
286   *          d3 - d2
287   *
288   *        bmax - bmin
289   * k4 = -------------
290   *          d3 - d2
291   *
292   * b = k3 + k4 da
293   */
294  return (BETA_MIN * d3 - BETA_MAX * d2 + (BETA_MAX -
       BETA_MIN) * da)
295    / (d3 - d2);
296 }
297
298 /* Update alpha and beta values once per RTT */
299 /* here again we added another paramter */
300 static void update_params(struct sock *sk, u32 adv_wnd)
301 {
302  struct tcp_sock *tp = tcp_sk(sk);
303  struct illinois *ca = inet_csk_ca(sk);
304
305  if (tp->snd_cwnd < win_thresh) {
306    ca->alpha = ALPHA_BASE;
307    ca->beta = BETA_BASE;
308  } else if (ca->cnt_rtt > 0) {
```

```
309     u32 dm = max_delay(ca);
310     u32 da = avg_delay(ca);
311     ca->alpha = alpha(ca, da, dm, sk, adv_wnd); // MKS
312     ca->beta = beta(da, dm);
313     //struct timeval ts;
314     //do_gettimeofday(&ts);
315           //s64 timestamp = (ts.tv_sec) * 1000000 +
                 (ts.tv_usec);
316
317     //printk(KERN_INFO "At time %lld: ADV = %u | CWND =
          %u | ssthresh = %u | MAX-RTT = %u | Min-RTT = %u
          | Sum-RTT = %lld | alpha = %u | Avg-delay = %u |
          Max-delay = %u\n", timestamp, adv_wnd,
          tp->snd_cwnd, tp->snd_ssthresh, ca->max_rtt,
          ca->base_rtt, ca->sum_rtt, ca->alpha, da, dm);
318
319
320                 /* MKS: increase the rounter counter */
321                 // ++ca->cnt_round;
322
323                 /* MKS: reset the base_rtt after 100
                       rounds */
324                 // if (ca->cnt_round > 100) {
325                 //    rtt_base_reset(sk, da);
326                 // }
327   }
328
329   rtt_reset(sk);
330 }
331 static void printValues(struct sock *sk, u32 adv_window){
332   struct tcp_sock *tp = tcp_sk(sk);
333   struct illinois *ca = inet_csk_ca(sk);
334   struct timeval ts;
335
336   //u32 dm = max_delay(ca);
337         //u32 da = avg_delay(ca);
338
339         do_gettimeofday(&ts);
340         s64 timestamp = (ts.tv_sec) * 1000000 +
                (ts.tv_usec);
341
342   //printk(KERN_INFO "At time %lld: ADV = %u | CWND = %u
        | ssthresh = %u | MAX-RTT = %u | Min-RTT = %u |
        Sum-RTT = %lld | alpha = %u | Avg-delay =  |
        Max-delay = \n", timestamp, adv_window,
        tp->snd_cwnd, tp->snd_ssthresh, ca->max_rtt,
```

```
        ca->base_rtt, ca->sum_rtt, ca->alpha);
343 }
344
345 /*
346  * In case of loss, reset to default values
347  */
348 static void tcp_illinois_state(struct sock *sk, u8
      new_state)
349 {
350   struct illinois *ca = inet_csk_ca(sk);
351
352         ca->curr_state = new_state; // Keep track of TCP
            CA state
353
354         if (new_state == TCP_CA_Loss) {
355           ca->alpha = ALPHA_BASE;
356           ca->beta = BETA_BASE;
357           ca->rtt_low = 0;
358           ca->rtt_above = 0;
359           rtt_reset(sk);
360         }
361 }
362 /* We send the adv_window as a parameter to
      differentiate between the two sub-flows */
363 static void prob_cwnd(struct sock *sk, u32 adv_wnd) {
364   struct tcp_sock *tp = tcp_sk(sk);
365   struct illinois *ca = inet_csk_ca(sk);
366         u32 adv_cwnd = adv_wnd; // this needs to r
367
368         // We should only reduce window when TCP is not
            in loss recovery mode
369         if (ca->curr_state != TCP_CA_Open)
370           return;
371
372         /*
373           When cwnd is bigger than the advertised cwnd
              and we are
374           also operating above the target delay we
              should set the cwnd to
375           the ADVERTISED cwnd.
376         */
377         if (tp->snd_cwnd >= tp->snd_ssthresh &&
378             tp->snd_cwnd >= adv_cwnd &&
379             ca->rtt_above >= 1) {
380           tp->snd_cwnd = adv_cwnd;
381           tp->snd_ssthresh = tp->snd_cwnd;
```

```
382         }
383 }
384
385 /*
386  * Increase window in response to successful
       acknowledgment.
387  */
388 static void tcp_illinois_cong_avoid(struct sock *sk, u32
      ack, u32 acked)
389 {
390   struct tcp_sock *tp = tcp_sk(sk);
391   struct illinois *ca = inet_csk_ca(sk);
392
393         /*
394           MKS: a window worth of data has been
                completed, time to
395           update params. The prob_cwnd function should
                be invoked
396           here to control queueing delay. It regularly
                cuts the
397             snd_cwnd to the advertised threshold.
398         */
399   //u32 ad_window = 0;
400   /*
401          * These if-conditions examine what are the
              subflows.
402          * subflow with IP of 192.168.10.0/24 will take
              Adv_CWND_1
403          * subflow with IP of 192.168.20.0/24 will take
              Adv_CWND_2
404          */
405         //if(((unsigned char*)&((struct inet_sock *)
              sk)->inet_saddr)[0] == 0xc0 &&
406         //    ((unsigned char *)&((struct inet_sock *)
              sk)->inet_saddr)[1] == 0xa8 &&
407         //    ((unsigned char *)&((struct inet_sock *)
              sk)->inet_saddr)[2] == 0x0a){
408        //            ad_window = ADVERTISED_CWND_1;
409         //}else if(((unsigned char*)&((struct inet_sock
              *) sk)->inet_saddr)[0] == 0xc0 &&
410         //         ((unsigned char *)&((struct inet_sock
              *) sk)->inet_saddr)[1] == 0xa8 &&
411         //         ((unsigned char *)&((struct inet_sock
              *) sk)->inet_saddr)[2] == 0x14){
412         //           ad_window = ADVERTISED_CWND_2;
413         //}else{
```

```
414         //          ad_window = ADVERTISED_CWND_1;
415       // }
416   if (after(ack, ca->end_seq)){
417     u32 ad_window = 0;
418           /*
419           * These if-conditions examine what are the
                subflows.
420             * subflow with IP of 192.168.10.0/24 will
                  take Adv_CWND_1
421           * subflow with IP of 192.168.20.0/24 will take
                Adv_CWND_2
422           */
423           if(((unsigned char*)&((struct inet_sock *)
                sk)->inet_saddr)[0] == 0xc0 &&
424             ((unsigned char *)&((struct inet_sock *)
                  sk)->inet_saddr)[1] == 0xa8 &&
425             ((unsigned char *)&((struct inet_sock *)
                  sk)->inet_saddr)[2] == 0x28){
426                   ad_window = ADVERTISED_CWND_1;
427           }else if(((unsigned char*)&((struct inet_sock
                *) sk)->inet_saddr)[0] == 0xc0 &&
428                 ((unsigned char *)&((struct inet_sock
                    *) sk)->inet_saddr)[1] == 0xa8 &&
429                 ((unsigned char *)&((struct inet_sock
                    *) sk)->inet_saddr)[2] == 0x3c){
430                 ad_window = ADVERTISED_CWND_2;
431           }else if(((unsigned char*)&((struct inet_sock
                *) sk)->inet_saddr)[0] == 0xc0 &&
432                         ((unsigned char *)&((struct
                            inet_sock *)
                            sk)->inet_saddr)[1] == 0xa8 &&
433                         ((unsigned char *)&((struct
                            inet_sock *)
                            sk)->inet_saddr)[2] == 0x14){
434                         ad_window = ADVERTISED_CWND_3;
435                 }
436     else{
437                   ad_window = ADVERTISED_CWND_1;
438           }
439
440                 update_params(sk, ad_window);
441                 prob_cwnd(sk, ad_window); // MKS
442         }
443   /* RFC2861 only increase cwnd if fully utilized */
444   if (!tcp_is_cwnd_limited(sk))
445     return;
```

```
446
447   /* In slow start */
448   if (tcp_in_slow_start(tp))
449     tcp_slow_start(tp, acked);
450
451   else {
452     u32 delta;
453
454     /* snd_cwnd_cnt is # of packets since last cwnd
          increment */
455     tp->snd_cwnd_cnt += ca->acked;
456     ca->acked = 1;
457
458     /* This is close approximation of:
459      * tp->snd_cwnd += alpha/tp->snd_cwnd
460      */
461     delta = (tp->snd_cwnd_cnt * ca->alpha) >>
          ALPHA_SHIFT;
462     //printk("delta %u and cnt %u  and clamp %u\n",
          delta, tp->snd_cwnd_cnt, tp->snd_cwnd_clamp);
463     if (delta >= tp->snd_cwnd) {
464       tp->snd_cwnd = min(tp->snd_cwnd + delta /
            tp->snd_cwnd,
465                  (u32)tp->snd_cwnd_clamp);
466       tp->snd_cwnd_cnt = 0;
467     }
468   }
469   //printValues(sk, ad_window);
470 }
471
472 // Let's use reno loss recovery instead of this one
473 static u32 tcp_illinois_ssthresh(struct sock *sk)
474 {
475   struct tcp_sock *tp = tcp_sk(sk);
476   struct illinois *ca = inet_csk_ca(sk);
477
478
479   //printk("-- window %u  and    betaa  %u and -----  %u
        ------------------ %u \n", tp->snd_cwnd, ca->beta,
        ((tp->snd_cwnd * ca->beta) >> BETA_SHIFT),
        max(tp->snd_cwnd - ((tp->snd_cwnd * ca->beta) >>
        BETA_SHIFT), 2U));
480   /* Multiplicative decrease */
481   return max(tp->snd_cwnd - ((tp->snd_cwnd * ca->beta)
        >> BETA_SHIFT), 2U);
482 }
```

```
483
484 /* Extract info for Tcp socket info provided via
      netlink. */
485 static size_t tcp_illinois_info(struct sock *sk, u32
      ext, int *attr,
486         union tcp_cc_info *info)
487 {
488   const struct illinois *ca = inet_csk_ca(sk);
489
490   if (ext & (1 << (INET_DIAG_VEGASINFO - 1))) {
491     info->vegas.tcpv_enabled = 1;
492     info->vegas.tcpv_rttcnt = ca->cnt_rtt;
493     info->vegas.tcpv_minrtt = ca->base_rtt;
494     info->vegas.tcpv_rtt = 0;
495
496     if (info->vegas.tcpv_rttcnt > 0) {
497       u64 t = ca->sum_rtt;
498
499       do_div(t, info->vegas.tcpv_rttcnt);
500       info->vegas.tcpv_rtt = t;
501     }
502     *attr = INET_DIAG_VEGASINFO;
503     return sizeof(struct tcpvegas_info);
504   }
505   return 0;
506 }
507
508 static struct tcp_congestion_ops tcp_illinois
      __read_mostly = {
509   .init    = tcp_illinois_init,
510         .ssthresh = tcp_illinois_ssthresh,
511         //.ssthresh       = tcp_reno_ssthresh,
512   .undo_cwnd  = tcp_reno_undo_cwnd,
513   .cong_avoid = tcp_illinois_cong_avoid,
514   .set_state  = tcp_illinois_state,
515   .get_info = tcp_illinois_info,
516   .pkts_acked = tcp_illinois_acked,
517
518   .owner     = THIS_MODULE,
519   .name    = "illinois",
520 };
521
522 static int __init tcp_illinois_register(void)
523 {
524   BUILD_BUG_ON(sizeof(struct illinois) >
        ICSK_CA_PRIV_SIZE);
```

```
525     return tcp_register_congestion_control(&tcp_illinois);
526 }
527
528 static void __exit tcp_illinois_unregister(void)
529 {
530     tcp_unregister_congestion_control(&tcp_illinois);
531 }
532
533 module_init(tcp_illinois_register);
534 module_exit(tcp_illinois_unregister);
535
536 MODULE_AUTHOR("Stephen Hemminger, Shao Liu");
537 MODULE_LICENSE("GPL");
538 MODULE_DESCRIPTION("TCP Illinois");
539 MODULE_VERSION("1.0");
```

Listing A.1: WhiteHaul Implementation in Linux Kernel