Home Network Interference Detection with Passive Measurements

Petter Juterud Barhaugen

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

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2023

Home Network Interference Detection with Passive Measurements

Petter Juterud Barhaugen

Home Network Interference Detection with Passive Measurements

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Issues in local area networks can be manifold, and most users lack the expertise or data needed to be able to deduce whether a problem arose in the local wireless network or elsewhere in the Internet path. This thesis introduces NETHINT – a novel tool that listens to wireless local area network traffic and collects long-term statistics such as delay and packet loss. These statistics can help pinpoint the origin of such problems. NETHINT presents them in a way that is easily parsable and exportable. Using reliable sources of metrics, it compiles a repository of accumulated data and lays a foundation for future statistic correlation methods.

Acknowledgments

Thank you to my supervisor, Michael Welzl, for the guidance and knowledge I needed to finish this thesis.

Thank you to my wife, Maria Juterud Barhaugen, who has supported and encouraged me throughout my entire time at the university.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network problems today are often obscure and difficult to diagnose for end-users. Whether experiencing slow transfer speeds, long delay times, or outright disconnection from their wireless Internet connection, it is easy to get frustrated not knowing what is causing them the trouble. During the COVID-19 pandemic, many people found themselves to be rather suddenly working from home. This situation brought a sudden increase in demand for high-quality, lag-free online communication for business meetings and the like to flow as normal. However, the cracks began to show as network problems became especially apparent during online video conference meetings. In fact, a 2020 study showed that 49% of respondents in Nordic countries had experienced Internet connectivity issues, and 25% experienced them at least once a week [7]. One can imagine that many of these issues arose during work hours. This problem is likely to not only occur in the present, but might continue to present itself in the future. Companies built solutions around home offices, and whether it is due to convenience, saving money, or the wishes of the company or employees, working from home has become more popular than ever before. All this underlines the fact that building reliable, cost-effective home networking solutions will only continue to be more critical.

When network issues arise, it is often easy to point the finger at other users on the local network, such as one's family members in the other room watching a movie, playing an online video game, or talking with their friends on video chat. Confirming the validity of these accusations is often difficult due to the lack of credible evidence. The truth is that for the majority of network connectivity problems, performance could be impacted by a myriad of factors at any point in the link between users and servers. It begs the question: when the rates of transfers exceed the path's capacity, how often, on average, are they bottlenecked at the access link of users in the building?

Many users will resort to investing in long-term solutions to try to mitigate local connectivity issues. One such solution is buying new, more expensive network equipment, *e.g.*, a new router, a Wi-Fi Access Point, or even a whole mesh Wi-Fi system. The costs of these measures are manifold. Not only are they expensive in relation to price, but they can also be a huge

time sink to install and configure. The needless replacement of perfectly fine hardware is a significant contributing factor when it comes to e-waste, not to mention that solutions like a mesh Wi-Fi system will only help the user if the problem at hand is a weak or unstable wireless signal. Some might even try upgrading their Internet subscription plan to accommodate bandwidths that far surpass their actual usage, hoping that it will give them enough in those moments when data transmission on the local network is at its peak. Another solution might be to change or configure the operating system of end-user devices. These solutions, too, are costly in time and money.

Instead of (or after) going through these measures, an end-user might want to put in some real effort to truly get to the bottom of what is causing issues in the network but could quickly run into problems. The traditional model for network monitoring is one where a user or administrator interactively runs tests and then views the result. For most end-users, such a model is probably not the most suitable approach to derive and present useful metrics and statistics, as it requires knowledge about the behavior of network protocols, packet loss, congestion, and the like. Furthermore, it might not be beneficial to anyone, regardless of their network competence level, considering the fact that running such tests would require the issue to persist for long enough for the user to run the tests themselves and collect all the related statistics.

The collected data might still not solve any of the user's problems. Web transfers often finish early, and long-term transfers, such as video streams, are typically bursty and do not behave like "greedy" flows (*i.e.,* flows constantly sending at their maximum rate). Finding patterns or apparent discrepancies in network quality might therefore prove to be difficult using this method. It would be interesting to see how often transfers experience a local bottleneck at all, and how much would be achieved overall by removing this bottleneck. Rather than looking at in-the-moment statistics, it makes sense that long-term measures, such as the ones mentioned above, would be supported by long-term statistics in terms of hours or days. Users could consult the measurement tool and, in hindsight, see the state of the network the moment they experienced issues.

A monitoring device separate from any endpoint of a connection would be able to see the activity of all connected devices on a local wireless network From this, it could infer various data points about latency, packet loss, wireless signal quality, and more.

In this thesis, we present NETHINT, a tool created to collect and parse these different statistical metrics about the state of the network and display them in a user-friendly way that allows users to get a good idea of network performance and use the data in diagnostics. The tool could potentially be used as a framework for correlating the collected data to estimate when the Wireless Local Area Network (WLAN) is acting as a bottleneck in the traffic flows of its connected devices.

Chapter 2

Background and Related Work

In order to use metrics to get an idea of the state of a WLAN, it is essential to know what those metrics are based on. In this chapter, we will introduce some key concepts such, as the underlying transport-layer protocols that form the backbone of the Internet as we know it. We will then discuss how these protocols and their mechanisms perform, including some of the various issues they currently face. Next, we will introduce the statistical metrics that might be relevant in assessing the state of a WLAN. By analyzing these metrics, we can gain a clearer understanding of a WLAN's performance. In addition, this chapter will explore the differences in measuring latency and throughput.

Finally, we delve into passive measurement techniques and tools. We will discuss how shared bottleneck detection could help us detect problems on a WLAN, as well as how NETHINT distinguishes itself from the existing tools.

2.1 Transport-layer protocols

The Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) are the two most widely used transport-layer protocols on the Internet. More recently, a newer protocol called QUIC has increasingly made up Internet traffic.

2.1.1 TCP

TCP is defined per its RFC document as "a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications" [51]. TCP enables much of the functionality that is essential to the Internet's operation today. The most basic functionality is providing the packet's source and destination port numbers, but it is the other fields in its header that define TCP. A figure of the TCP segment header can be viewed at Figure 2.1.

The sequence (SEQ) number, as the name implies, indicates the current segment number in the sequence of bytes in the flow. The Sender Maximum Segment Size (SMSS) dictates the segment size. The sequence

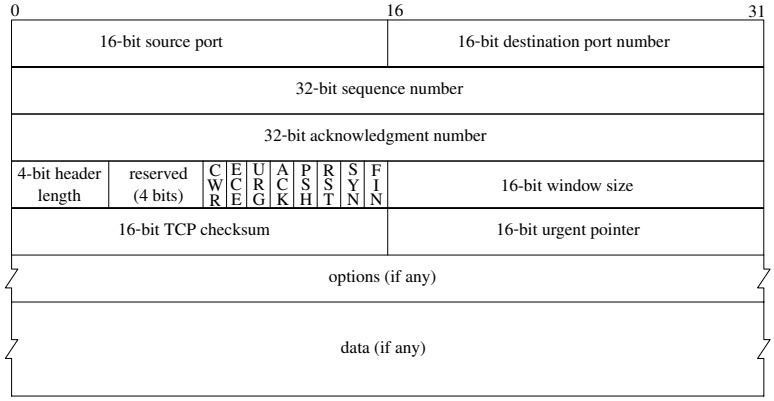


Figure 2.1: TCP segment header. Taken from [46] and updated to reflect the most recent TCP specification [9].

number is helpful as the receiver of the packet can count the subsequent packets' sequence numbers in order to validate that they have received them all.

It can also notify the sender that it received the packet by setting the acknowledgment (ACK) bit in the header and mirroring the received sequence number in the acknowledgment number field. ACKs carry the number of the next expected byte to be received. If packets with out-of-order sequence numbers are received, the receiver can send a Duplicate acknowledgment (DupACK) packet with the same ACK number as the previous one. These DupACKs are a way to detect packet loss, described in more detail in Section 2.3.2.

Another way TCP does error control is with timeouts. It keeps track of a timeout variable called the Retransmission Timeout (RTO), which is adjusted based on the connection's Round Trip Time (RTT). The RTO expires when TCP has received no ACKs in the specified period, and as such, TCP will lower its sending rate and start sending retransmissions. Balancing the RTO can be a complex problem, as too-low values would lead to unneccessary retransmissions, and too-high values would mean slow reactions to packet loss. Retransmissions are discussed further in Section 2.3.2

A delayed ACK is a way for TCP to improve network efficiency by waiting for a short delay for multiple incoming packets before ACKing them. It can potentially lead to higher latency if one end of a connection stops transmitting packets and the other has to wait for a timeout to send an ACK. There can also be additional delays, based on the method of measurement, which are further discussed in Section 3.1.1.

TCP incorporates a three-way handshake to establish a connection by exchanging SYN and SYN/ACK packets, and finally, an ACK packet which can optionally include data. It later goes through the same process with FIN and FIN/ACK packets to terminate the connection.

2.1.2 UDP

UDP, as opposed to TCP, is a so-called "connectionless" protocol, meaning it does not set up an end-to-end connection, has no handshake process, and has no way of knowing if sent packets arrive at the other end. In addition to source and destination numbers, its header only includes a length field (made redundant by the length field in the underlying IP header) and a checksum (optional in IPv4). UDP's advantage over TCP is that it is fast, and requires no handshake to set up a connection. It simply sends packets until there are no more packets to send, making it optimal for real-time applications such as video games, where latency matters and there is no time for retransmission.

Due to its lack of sequence numbers and connectionless nature, it will not work with any congestion control mechanism.

2.1.3 QUIC

Despite the unadorned specification of UDP, a newer transport-layer protocol named QUIC utilizes UDP by encapsulating all packets in datagrams. It implements a connection-oriented protocol with both congestion control and automatic retransmission. As network devices have come to almost exclusively expect TCP or UDP-type transport-layer packets, QUIC needed to be implemented in the application layer (see discussion of "ossification" in Section 2.5).

QUIC is a multiplexed protocol, with support for 0-RTT TLS (Transport Layer Security) handshake, meaning it can send application data along with the opening packet, saving a round-trip [49]. Multiplexing means it mitigates the problem of Head-of-Line (HoL) blocking, where packets must wait for all packets with lower sequence numbers to be ACKed before getting delivered. This problem is common to TCP, which handles all data as a strictly in-order byte stream. What is commonly referred to as HTTP/3 is, in fact, HTTP over QUIC, as opposed to the conventional TCP or UDP.

QUIC encrypts its payload as well as its header, for the sake of user privacy, leaving only some flags and a connection ID in cleartext, thus obscuring the sequence and ACK numbers [27]. There has been some debate around whether this might hinder research and development of network protocols, as researchers often rely on the open data in transport headers. Although most of the QUIC header is encrypted, there is one unencrypted flag called the "spin bit", which is simply inverted once per RTT. During testing for this thesis, the spin bit was never enabled in the encountered QUIC packets. It is also prone to failure under packet reordering.

On the other side of the debate, encrypted transport headers might serve to minimize transport layer ossification as network devices cannot rely on strict header structure conventions [10].

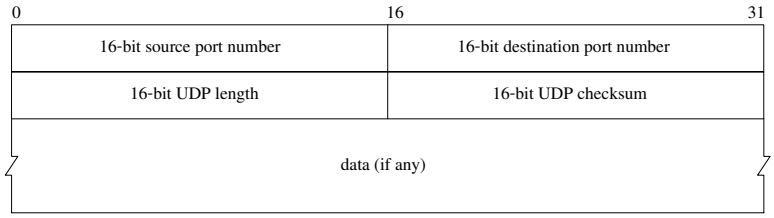


Figure 2.2: UDP datagram header. Taken from [46].

2.2 Congestion control

Congestion control is the mechanism within TCP that decides the rate at which packets can be transmitted in order to not *congest*, or exceed the available bandwidth on a connection path between two nodes [52].

2.2.1 Development of congestion control

It was first proposed in the late 1980s to mitigate a "congestion collapse", where packets would be dropped or severely delayed at choke points where traffic exceeded the available bandwidth. The "congestion window" (CWND) was introduced, which dictated the number of packets the sender would transmit each RTT. The CWND was based on an idea called ACK clocking, which means to transmit segments upon the arrival of the ACKs. This window would increase until it hit the maximum capacity of the path (indicated by a packet loss), then TCP would reduce the CWND and start the process over.

This approach would go on to become the standard for a couple of years. However, in the 2000s, TCP itself became a bottleneck as the CWND in high-capacity links would increase so slowly that TCP took an exceedingly long time to fully saturate the potential capacity, if it ever got to do so at all. Transfers would often not last long enough for this to happen.

There have been many implementations of congestion control over the years in an effort to solve this problem. The first implementation was called TCP Tahoe, which employs a simple scheme. The Initial Congestion Window (IW) is first set to a certain number of data segments. The IW is then exponentially increased until packet loss occurs, in a process called TCP slow start. It then enters a new phase called congestion avoidance, where it gradually increases its CWND until it detects a packet loss because an RTO occurs. Then, the process starts anew with the IW[1]. Tahoe was later improved upon with TCP Reno, which will halve the CWND and go right into the congestion avoidance phase instead of returning to the IW for every packet loss.

In the end, an implementation dubbed "CUBIC" was added to the Linux kernel as the default congestion mechanism. It was designed to increase the transmission rate faster than existing congestion control protocols, but back

[1]In fact, depending on implementations, this value is limited to 1 or 2 packets, which can be significantly less than IW today.

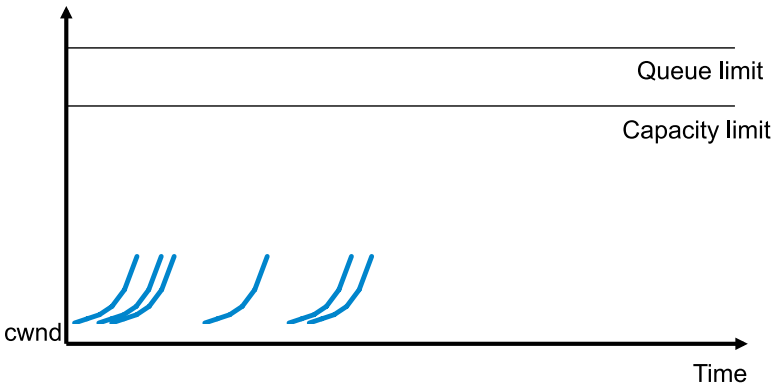


Figure 2.3: Illustration of connections attempting to probe for capacity, but being too short to do so. Taken from [53].

off enough to make room for other connections in times of high packet loss rates.

2.2.2 Congestion control today

Today, we face a similar problem as researchers in the early 2000s. We have an abundance of bandwidth across Internet links, but they are rarely saturated. CUBIC is designed for long-lasting connections with a great deal of data to transfer, but only a few such connections are being made. Web pages are small and are quickly transmitted (see Figure 2.3), and even as video streaming, such as Netflix, has become a significant portion of Internet transfers, they are usually broken up into short bursts of data [1]. This means they are rarely able to probe for available bandwidth for long enough to increase their congestion windows to their optimal sizes. In fact, one 2022 study showed that 85.5% of all Internet flows measured do not have a single packet loss for the entire duration of the connection, which speaks to how infrequently they reach their full capacity potential [53]. This problem becomes bigger yet when considering wireless transfers. If a connection should be terminated because of an unstable signal strength, the congestion window will still reset, meaning it has to restart the process of probing for capacity.

Most standards set the IW to 10 data segments, with some servers, such as Content Distribution Networks (CDN), setting it to other specific values to suit their needs [39]. It is then exponentially increased in the TCP slow start process. It might be easy to suggest a larger IW or a more aggressive slow start, but it is important to remember that there are still slower links around the world. In fact, one study showed that old, slower links (especially in poorer countries) do not disappear nearly as quickly as new ones appear [53], as illustrated in Figure 2.4. Large IWs may lead to bursts of heavy congestion on links that are not equipped to handle a sudden increase in data. All this suggests that the static congestion control mechanisms that are in place today could stand to benefit from being more dynamic; *i.e.*, having several points of feedback within one connection flow

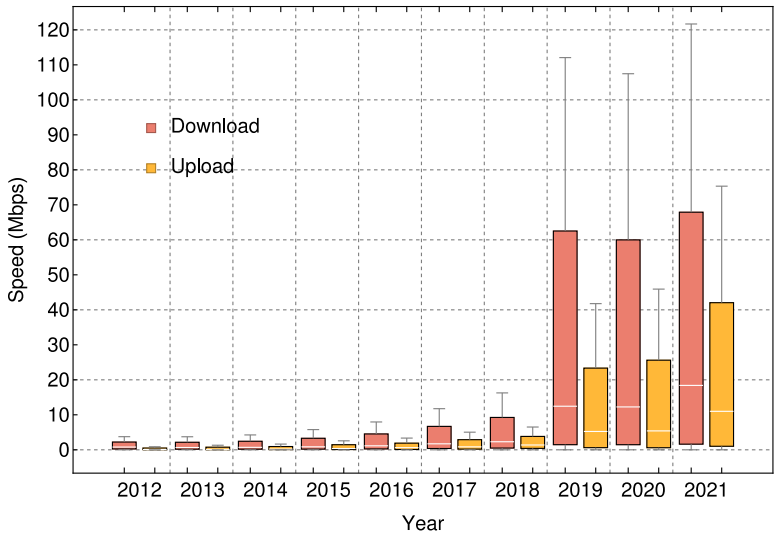


Figure 2.4: Throughput ranges in the Internet over the years. Taken from [53].

instead of one at either end. Internal control loops such as these may give a more accurate look at the capacity along the links in the connection.

The concern of state-of-the-art congestion control mechanisms is of interest, as the data would indicate that, more often than not, the experienced Internet problems of users are happening due to either congestion- or application-specific limitations.

2.2.3 Bufferbloat

Bottlenecks in the network commonly stem from over-full packet queues at edge- or intermediary nodes. When these queues (or buffers) fill up, they will begin to drop packets. However, many hardware vendors have increased buffer capacities in intermediary nodes to such a degree that nearly all packets are stored and acknowledged, thus never dropping packets. While this might in theory sound like a positive development, packet loss remains the primary signal which triggers congestion control. Senders that never receive signals to slow their sending rate will keep increasing speeds until the buffers at the intermediary nodes grow exceedingly large and negatively effect the connections' RTTs. This phenomenon is known as "bufferbloat", and is being actively researched by groups such as The Bufferbloat Community [6]. For different categories of traffic, such as video streaming and conference calls, Active Queue Management (AQM) could ensure stability for both connections. Proposed solutions such as FQ_CoDel [19] (Fair Queuing Controlled Delay) have shown to improve delay times by utilizing AQM, but are not implemented widely enough that bufferbloat can be considered a solved problem. Even though end network devices such as routers or servers are often known to have large buffers, a bottleneck due to bufferbloat could happen anywhere on a given network path. The proposed tool could help inform whether

the problem exists on the WLAN, and if there is anything a user could potentially do to mitigate the experienced issues. It could also potentially inform future fair queuing research on where to focus their efforts; on network edges or other nodes along the path.

2.3 Statistical metrics

In order to assess the state of the local network, some key statistical metrics are required to be collected and processed. There more of these metrics are collected and presented in a user-friendly way, the better a picture is painted of the WLAN.

2.3.1 Packet Delay

The RTT is the measure of time it takes a sender to send a packet until it gets a response to said packet. In congestion control, RTT represents how often a mechanism can probe for capacity, and thereby signifies the "aggressiveness" of the mechanism. Okada et al. [34] observed that TCP acknowledgment packets are often received after a delay in the WLAN layer, meaning higher RTTs and subsequent TCP performance degradation. Delay, or latency, directly impacts the user experience, especially in real-time applications such as video communication or online gaming. As such, it is vital to measure RTTs to get a better picture of the network performance.

2.3.2 Transport-layer retransmissions

The number of transport-layer retransmissions a connection experiences indicates packet loss, and therefore a potential bottleneck somewhere along the connection path.

Defining a metric that indicates when a packet fails to reach its destination is necessary to measure latency on a local network and enable congestion control. This metric is packet loss, and it can be categorized into three possible cases.

The first case is when loss happens before the point of measurement in our network. In order to detect this, the sequence numbers in the packet headers can be analyzed. A retransmission is detected to have occurred if a sequence number occurs twice in a row, and from this, packet loss can be inferred.

The other case of packet loss is when it occurs after the measurement point. To detect this, we can look at duplicate ACK packets. If three duplicate ACKs are received in a row, it indicates that a packet was lost [45].

The third and final case of packet loss is dubbed "tail losses", which happens when the last data packet in a connection, or a "FIN" (connection termination) packet, is lost before the measurement point. TCP handles with the RTO timer mentioned in Section 2.1.1. Measurements have shown that on Google servers, about 70% of all retransmissions are sent

after the RTO expires, as opposed to being handled by a fast recovery mechanism [8]. These timeouts are especially costly for short flows, as such flows are often interactive, and low latency is critical. Waiting for the RTO to expire at the end of a connection can be detrimental to a user's experience.

Even on stable networks, there can be significant packet loss when flows converge to a single point. The hardware buffers may not have sufficient capacity to hold all the packets before forwarding them, resulting in the need to reject, or drop, new packets.

2.3.3 Explicit Congestion Notification

The last metric is the Explicit Congestion Notification (ECN) which is a flag sometimes enabled by setting the two ECN bits in IPv4, or the two right-most bits in the Traffic Class field in IPv6. Instead of simply dropping a packet and thereby signaling the sender, this flag allows a receiver to keep the packet and instead notify the sender explicitly that its buffers are full. Being in the IP layer allows it to be enabled for all protocols above it, but unfortunately, the upper layer needs to handle setting it [12]. It needs to be enabled on both ends of the connections and the underlying network structure, and is not enabled on most web servers ($< 1\%$ in 2014) [50]. The limited amount of connections it is enabled for means that NETHINT does not check ECN.

2.3.4 Wireless performance metrics and challenges

Signal quality and sending rate are two crucial metrics in Wi-Fi. Understanding their relationship is critical, as these metrics explain the cause when a bottleneck occurs in a WLAN. Being a wireless physical medium, it is prone to many added challenges that do not occur in other physical media such as Ethernet or fiber-optic. These challenges are discussed after the metrics.

Wireless signal quality

The assumption that a bottleneck's capacity is stable does not hold anymore. Along with the emergence of millimeter wave (mmWave) 5G wireless (IEEE 802.11ay) and novel local wireless standards such as Wi-Fi 6 (IEEE 802.11ax), the chance of a wireless signal making the path more unstable is more significant than ever. Handoffs or temporary wireless instability can lead to spurious timeouts and long delay times.

Collision is a term used to describe the event that occurs when two or more stations attempt to transmit data over the same channel at once, and is a contributing factor in packet loss and, thereby, congestion. It was solved in Ethernet with switches, which allows for communication that is free from the collision potential still present in the wireless space.

In the wireless medium, the stability and transmission rate of a connection depends on the spatial locality of the communicating devices in

relation to each other. In the past, Wi-Fi transmitted data omnidirectionally, meaning the conditions of a connection would quickly decrease when moving the receiving device away from the AP. Using beamforming, Wi-Fi 6 devices can direct their frames at devices at different spatial locations around them, and in fact, combining beamforming with MU-MIMO makes it possible to send to eight spatially diverse devices (dubbed *beamformees*) simultaneously [23]. This MU Beamforming is very time-dependent as all of these devices may move around, requiring fast and frequent channel measurement to stay up to date on where to direct the signal, and as such, the beam can be changed on a per-frame basis [14]. Above a certain distance, however, the effectiveness of beamforming between devices is insignificant. Despite these improvements, collisions cannot be ruled out and are still a contributing factor to poor connection performance. Wired connections have yet another advantage as Ethernet cables can keep their rates up even over 100 meter distances [21].

Signal interference is another worry with wireless. Other wireless networks in the area, or even household objects such as older microwave ovens and TVs, may cause electromagnetic interference in the bandwidth range of Wi-Fi, worsening the signal. It is the signal strength and noise which inform the sending rate.

Sending rate

In wireless networks, the sending rate is a product of the signal quality. The sending rate of an Access Point (AP) marks the frequency at which the wireless medium can transmit data to receiving nodes. Before dispatching, data is put into frames which might be aggregated together for efficiency. It makes intuitive sense that the sending rate should be as high as possible, but it is sometimes favorable to limit it. High rates combined with a poor signal lead to a higher probability of packet loss, meaning the need for retransmissions that occupy the channel and block other connections.

It is worth looking into this metric since wireless throughput suffers when the frame transmission rate or the frame size decreases. Different Wi-Fi protocols support different sending rates, which are mainly decided by the modulation schemes in use by the transmitter. These modulation schemes are represented in wireless packet headers as a Modulation Coding Scheme (MCS) index.

In order to achieve high throughput, the rate of transmissions are fine-tuned repeatedly in case the channel conditions change. One widely deployed rate control mechanism is Minstrel, which periodically calculates the maximum throughput possible. It calculates throughput as

$$TP = P_{new} * \left(\frac{1sec}{SIFS + T_{DATA}} \right) * FrameSize \tag{2.1}$$

where P_{new} is the probability of success of the current rate adaptation window, $SIFS$ is the Short Interframe Space (the time it takes to process a wireless frame and return it), and T_{DATA} is the transmission time of the data frames [55].

Sending rates sometimes need to be limited as transfers at high rates combined with poor signals have a higher chance of packet loss, and throughput will suffer as a consequence. It is, therefore, beneficial to consider the sending rate of the wireless medium, which can be done by sniffing 802.11 MAC-layer frames.

Lower capacity

A challenge with Wi-Fi is that the capacity is lower in the wireless space than in the wired, despite recent advancements in wireless communications technology. Wi-Fi 6 utilizes OFDMA (Orthogonal Frequency Division Multiple Access) and higher bandwidths. This allows it to theoretically achieve tremendously high throughput under ideal conditions. It also uses Multi-User (MU) MIMO (Multiple-Input, Multiple-Output), which uses multiple transmitters to serve multiple users simultaneously over individual spatial streams. Among the advantages of this is the fact that several of these streams can be used to transmit data to a single device, leading to a higher capacity dependent on the AP and device's MIMO support, effectively multiplying the throughput with the number of streams in use [37]. However, despite these recent improvements in wireless capacity, it does not yet measure up to its wired counterpart. Ethernet cables have been capable of 10 Gbit rates for decades now [22].

802.11 Performance anomaly and airtime fairness

Apart from the issue of bufferbloat, one of the most detrimental performance issues in Wi-Fi is a phenomenon known as the 802.11 performance anomaly. The MAC protocol was created to guarantee throughput fairness between hosts on a WLAN by making all the hosts lower their sending rate to match that of the host with the lowest rate. This was discovered to have severely impact the performance of said WLAN, as one host far away from the nearest AP would limit every other host [16].

There have been many proposed solutions to this problem, but they have yet to be widely implemented. However, one potential implementation based on airtime fairness has made it into hardware such as the NetGear NightHawk routers [20]. This solution divides sending time fairly across the hosts on the WLAN, which allows devices with fast sending rates to co-exist with slower (and perhaps older) devices. [18].

It is vital to know about this issue. If a user's AP does not support any airtime fairness protocol, then a single device transmitting at a low data rate could create a bottleneck on the WLAN. All the other hosts would then also lower their rate, decreasing overall performance.

2.4 Latency vs. throughput

Internet capacity, or bandwidth, has become the de-facto way to measure the "speed" of an Internet connection. For years now, however, it has become apparent that another form of measurement is needed to improve

Internet performance, namely latency, also known as delay. Despite its importance in inter-network communication, latency has been largely neglected in favor of throughput (capacity over time), which has been the main focus for network infrastructure and application development. As a result, it is now arguably considered the biggest obstacle in inter-network communication.

Altering this perception for network equipment and infrastructure companies, as well as Internet users, is a crucial step in solving this problem. Some of the world's largest companies are showing great interest in pursuing an easy-to-understand way of collecting and presenting network connectivity metrics. Apple has recently introduced its own latency-based metric called RPM (RTTs Per Minute) in order to increase user comprehension of delay times with an intuitive "bigger is better" benchmark [35]. Committees such as the Internet Architecture Board have held workshops focusing on the issue of latency, citing the fact that one possible reason for the lack of development in this space is the common misconception that throughput matters over latency for a good Internet connection [25].

2.5 Passive measurement: approaches and tools

A measurement tool that monitors the network and looks for packet loss should do so passively. This entails simply listening to and deriving information from existing packets rather than actively probing the network for information. Existing active measurement techniques such as *ping* insert ICMP (Internet Control Message Protocol) packets into the network designed to measure the RTT of the connection path. There are a couple of reasons why this is undesirable. First, routers and switches along the connection path may handle such ICMP packets differently, leading to unrepresentative latency measurements. They also only inform about the delay on the network path during the period they are being transmitted, which is not very informative in the longer term. Inserting additional packets into the network adds extra load that can influence performance [57].

Another way of measuring RTTs is manipulating packet contents to insert specialized data such as identifiers or timestamps. However, this may also affect performance since it requires an additional processing step for each transmitted packet.

Another advantage to doing passive measurements is that there is no need to update routers or protocols with new software or mechanisms. A significant reason why novel network mechanisms are slow to be implemented as a standard is "ossification"; the notion that the more widely deployed a protocol is, the more difficult it is to change [36].

The measurement tool will see everything from the applications' point of view, which is what matters when it comes to looking at the behavior of the network.

2.5.1 Packet capturing techniques

In order to measure latency in the local network, one needs to look at the packet loss rate. Capturing packets on a network interface card is quite a low-level operation, but luckily, there are ways to monitor network activity with little difficulty. By default, it is only possible to capture unicast packets (packets intended for the host the measurement tool is currently running on), but it is possible to circumvent this. Setting the network interface to monitor mode enables it to pick up raw 802.11, multicast and broadcast packets, thereby capturing all packets on the Wireless LAN (WLAN).

2.5.2 Shared bottleneck detection

Our goal with this setup is to be able to see whether or not a bottleneck exists on the WLAN, thereby denoting a shared bottleneck for all the devices that are wirelessly connected to it.

There are several benefits to detecting such a shared bottleneck. As mentioned in the introduction chapter, knowing whether a bottleneck exists on the WLAN can potentially inform users on whether or not they need to upgrade existing hardware or Internet bandwidth subscriptions.

When it comes to detecting a bottleneck, it is necessary to determine the relevant data metrics available for analysis, and the primary metric to consider is packet delay. Looking at the latency times of packets can tell us a lot about the state of the network at a point in time when the monitoring tool has been active and receiving packets. Two or more connections' simultaneous drop in connection speed, *i.e.,* a spike in latency, is likely the result of a shared bottleneck on the common WLAN, something which is discussed further in Section 4.1.1.

Another metric is packet loss, which is simple enough to detect with little margin of error. However, it does not occur often enough in Internet traffic that it should be considered the most important. It is still worthy of inclusion, though, especially if a big number of losses take place in a short time frame.

Rubenstein, Kurose and Towsley [38] proposed both a loss-based correlation technique and a delay-based correlation technique. As packet loss is a relatively rare occurrence on the Internet today, it is most beneficial for us to look most closely at a delay-based technique. The paper bases itself upon two important points, the first being that if two packets pass through the same point of congestion, they should display some degree of correlation, *i.e.,* delay or packet loss, decreasing as the period between packets increases. The second point is simply that if two packets do not share a point of congestion, they are unlikely to show any degree of correlation.

There are several other reasons to perform shared bottleneck detection. Congestion control algorithms such as LEDBAT (Low Extra Delay Background Transport) are designed to be "lower-than-best-effort" (LBE) and perform tasks such as downloading software patches in the "background", disrupting traffic as little as possible [43]. When there is no shared bottle-

neck, these algorithms have minimal effect, as they might as well send at a much higher rate. Knowing whether or not such a bottleneck exists could therefore help inform LBE congestion control algorithms such as LEDBAT that they can be more or less aggressive, or to help choose the optimal congestion control algorithm in multipath transfers [15].

In Multipath TCP (MPTCP), a connection can use multiple paths simultaneously in order to improve the efficiency of the network. Nevertheless, when a bottleneck occurs, it does not differentiate on which path it did so. This motivates conservative behavior from the congestion control mechanism in use, as it will apply the same limitations to the uncongested path as it will to the congested one. As a result, there have been efforts to create Shared Bottleneck Detection (SBD) algorithms to identify which paths have bottlenecks in a connection and adjust the congestion control accordingly. One implementation, named MPTCP-SBD, even observed throughput gains of up to 40% with two subflows and more than 100% with five subflows [11].

It is worth keeping in mind that finding bottlenecks in the Wide Area Network (WAN) is not a solved issue, and may therefore impact the finding of bottlenecks in the Local Area Network (LAN).

2.5.3 Tools

This section will provide a detailed overview of the tools used in this thesis, including their functionalities, benefits, and limitations. The following tools were either critical for data collection and implementing NETHINT or somehow inspired it.

TCPDUMP

tcpdump is a program developed by a network research group led by Van Jacobsen in 1988 [30]. It was made at the time to have a simple way to look at packet traces in order to understand and fix issues in existing communication protocols. However, it ended up as the de-facto standard for command-line packet analysis. It uses a filtering technique called the Berkeley Packet Filter (BPF) in order to translate high-level filter descriptions into driver-level bytecode [31], and allows the user to save the resulting packet capture into a "pcap" file (pcap being an abbreviation of packet capture).

LIBPCAP

The researchers wanted to expand out from tcpdump and create other packet capturing applications but had already done the work of compiling high-level filters into bytecode with BPF. They decided to extract functionality from tcpdump and create a portable library called libpcap.

It is very low level, meaning it communicates directly with the network interface. However, it also means that it is quite a lot of work to get it working, and it is not very portable.

The tools Wireshark and tshark, mentioned below, also have the ability to capture packets and write pcap files using the newer pcap file format pcapng, which has extended functionality.

Scapy

Scapy is a Python module primarily focused on crafting and manipulating network packets of various protocols, but it is also capable of sniffing and parsing packets passively. Its relatively simple implementation makes it ideal for the tool described in this thesis, and writing it in Python makes it far more portable than if written in C, using purely libpcap.

Another advantage of writing NETHINT in Python is the interoperability with modern frameworks for simple, clean user interfaces that are easy to set up and access, as well as graphing and other data visualisation tools. It automatically selects which network interface to listen to, but it can also be set manually.

Wireshark and tshark

Wireshark and its command line companion tool, tshark, are tools made to capture live network traffic or read packet capture files, dissecting and displaying the information contained in the observed packets. They are simple to install and use for their purpose, but do not give an easily understandable overview of the state of the local network. Wireshark does include a way of viewing RTTs, but they are not necessarily basing them on reliable delay sources (see Section 3.1.1).

PPing

Pollere Passive Ping (PPing) is a tool for passively measuring the delay between two endpoints from any point in the connection path. Similar to *ping*, it reports the RTT of a connection, although with much higher resolution and without actively injecting additional traffic into the network. It is also not limited to run on an endpoint. It matches every packet containing an unseen timestamp value in its TCP header with the first packet in the reverse direction with the corresponding timestamp echo reply. It then calculates the time between the packets that were observed [33].

The method of inferring packet delay times from the TCP header's timestamp option is discussed further in Section 3.1.1.

ePPing

ePPing (evolved PPing) is a novel tool building on the existing PPing theory and extends it to run almost entirely in kernel space using eBPF (evolved BPF). Existing tools such as PPing rely on packets cloned from kernel space into user space in order to process them, introducing a significant overhead in terms of performance. Using new features in

the Linux kernel, a recent study was able to utilize various hooks into the kernel to process packet headers in kernel space, only sending the resulting RTTs to user space for displaying [48]. They observed that their tool processed around 18 times as many packets as PPing with 1000 flows. PPing was shown to only process about 6% of packets for 10 concurrent iPerf3 flows as it could not keep up with all the traffic, whereas ePPing processed 100%.

Wireless Diagnostics

MacOS's "Wireless Diagnostics" application is an existing tool that resembles the envisioned new tool to some degree; it shows quite extensive network metrics but is lacking in some regards. First and foremost, it does not differentiate between different applications or devices, showing only the network usage of the current host and not any of the other machines on the Local Area Network (LAN). This means that while it can observe the fact that the network *was* congested at a point in time, it cannot see exactly *what* caused the congestion. Gaining this insight would possibly clear up what is congesting the network, and give us more of an idea of where to direct our focus.

It also only shows real-time statistics and does not enable the user to go back and view the status of the network at a certain point in time. These factors make it difficult to obtain and compare network usage statistics between the hosts, giving us very little knowledge as to whether a bottleneck has occurred on the LAN.

2.5.4 Distinguishing features of NETHINT

NETHINT offers distinctive features to enable an interactive, user-friendly way of viewing accurate statistics calculated from data in the collected packets. A fundamental aspect of our tool is the fact that it only does passive measurements and does not interfere with the network in any way that might have a detrimental effect on the quality of experience for any user. Some other tools, such as ping, inject packets into the network and might receive inaccurate delay estimations and, at worst, disrupt the network service.

Many of the aforementioned existing tools consider the latency between hosts on the WLAN and the AP. Our tool is made to be placed on a device close to the hosts, disregarding the intra-network delay in favor of reduced complexity. This is possible since the capture time of the packets going past the monitoring device and the end hosts is so short, allowing NETHINT to see traffic as experienced by the devices on the WLAN.

Similar tools are often created to only look at the flows of traffic going in and out of the NIC of the host they are running on, limiting what they are capable of inferring about the WLAN as a whole. NETHINT, on the other hand, is created to run as a monitor for the entire WLAN, meaning it can compare the flows of different devices and potentially find out whether a bottleneck resides locally or somewhere else on the path.

	ping	PPing	ePPING	Wireshark	NETHINT
Passive measurement		X	X	X	X
Multi-device network statistics				X	X
Plots collected statistics				X	X
Collects multiple metrics (RTT, OWD, RSSI...)				X	X

Table 2.1: An overview of the features of NETHINT vs. other existing tools.

Contrary to tools such as Wireshark, which collect and display as much data as possible, NETHINT will only collect and parse what is necessary to give the user of the tool enough data to get a sense of the overall state of their WLAN at a certain point in time. For example, by filtering only valid RTTs to be displayed, the user is presented with more accurate and easily readable data than similar tools, which can be hard to navigate if they do not know specifically what they are looking for. This data includes multiple metrics that other tools may not collect. The data can then be logged to a human- and machine-readable JSON file or viewed in an interactive GUI. A table with an overview of the features in NETHINT compared to other existing tools can be seen in Table 2.1.

Chapter 3

NETHINT Design and Implementation

This chapter focuses on the implementation of NETHINT, and is divided into three sections. The first section focuses on statistical metrics, how they are gathered from various sources, and what metrics are considered reliable or unreliable.

The second section goes into detail about the tool itself, what it is, how to use it, and how it is implemented. This includes what is required to run it, as well as how to export data out of it for later analysis.

The third and final section details the monitoring hardware used in this thesis, and what is required to capture the relevant data.

3.1 Measurement and application of metrics

There is a myriad of tools and techniques for measuring all the metrics listed in Section 2.3, some of which are already mentioned in Section 2.5.3. *ping*, packet manipulation, and injection of bespoke measurement packets into the network are all active measurement methods. Since the goal is to measure passively, we must deduce these metrics from the pre-existing information in packet headers and other sources, which are discussed in this section.

3.1.1 Inferring delay from different sources

Not all ways of measuring packet delay are dependable enough to inform NETHINT about the RTT on the network path. We classify various sources of delay times as either reliable or unreliable based on their consistency in producing accurate results. The reliable sources can be trusted to yield accurate delay measurement data, consistently being a good indicator of what is realistically the state of the network.

Reliable delay sources

When considering TCP connections, what data can be considered reliable is dependent on whether or not the server has a high chance of immediately responding to a packet, so that the measured time does not include additional buffering unrepresentative of the RTT on the link or between applications on either side of the connection. Since we disregard the delay between the measurement point and local hosts, it is easy to infer delay times from outgoing packets, as one can simply look at the time delta between the packet being sent and a reply to that specific packet. With incoming packets, however, it is not necessarily known at specifically what time the packet was sent, nor when our reply arrives at the other end. Table 3.1 contains an overview of what are considered by us to be reliable delay sources of outgoing packets.

It is important to note that *only* the packets ACKing outgoing packets which carry data (apart from SYN or FIN packets, discussed below) is considered a reliable source of delay times. Outgoing "pure ACKs", *i.e.*, packets marked as ACKs without any payload, do not automatically invoke a response from the other end of the connection and can therefore not be counted on to give reliable delay times.

The foundation of our method of gathering reliable delay times is simple capture time, which is a straightforward measure of when data packets are received by devices on the WLAN or, more specifically, when the monitoring device receives them. This metric is particularly useful because our monitoring device is located in close spatial proximity to the receivers, meaning that the time when a packet passes by the monitoring tool is almost the same as when it gets to the receiver. Additionally, capture time is quite granular, measuring in microseconds, which allows us to capture even slight variations in packet delivery times. Combining the granular capture time with reliable methods of pairing outgoing data packets with incoming ACKs and inferring delay times from them results in accurate RTT estimations of Internet connections.

After finding a valid packet pair, the RTT can be simply calculated as:

$$\text{RTT} = \text{Capture\_time}_{\text{ACK}} - \text{Capture\_time}_{\text{Data}} \tag{3.1}$$

For incoming TCP packets, connection establishment and termination packets can be used for both packets with timestamps enabled and without, as described below. In addition, for timestamp-enabled packets, the One-Way Delay (OWD) from the server can be used as an indication of congestion. As clock synchronization can not be guaranteed or even expected, the timestamp values from the sender can not be directly compared to the local clock. Nevertheless, a relative OWD can be calculated by subtracting the TSval of a received packet from the capture time of the packet using the clock on the monitoring device. The resulting number is not indicative of anything by itself, as it is simply the difference between the monitor device's clock and the TCP timestamp clock of the sender plus the OWD. It can be used for comparison to previous packets in the same flow. If $OWD_{current} - OWD_{previous}$ increases between packets, it is

Delay source	Inputs
TCP SEQ/ACK based pairing	• RTT of outgoing data packets • Connection establishment packets • Connection termination packets
TCP Timestamps based pairing	• RTT of outgoing data packet retransmissions • Connection establishment packets • Connection termination packets
QUIC	• Initial and handshake • Spin bit

Table 3.1: Reliable delay sources of ***outgoing*** connections

Delay source	Inputs
TCP SEQ/ACK based pairing	• Connection establishment packets • Connection termination packets
TCP Timestamps based pairing	• OWD from server • Connection establishment packets • Connection termination packets
QUIC	• Initial and handshake • Spin bit

Table 3.2: Reliable delay sources of ***incoming*** connections.

a sign of congestion on the path. This is possible because the difference in the clocks is, for the most part, constant.

Clock skew is another issue worth mentioning. According to Zander and Armitage [56], clocks usually only skew about a few hundred ppm, or around 0.1 ms for an RTT of 100 ms. Combined with the fact that the measurement periods in our tool are both small and frequent, the possibility of clock skew becoming a significant concern is effectively mitigated. The previously mentioned congestion control algorithm LEDBAT employs a similar scheme to get a delay estimation, calculating a "base delay" from the sum of constant delay components [43].

The monitoring tool sets the base delay to be that of the previous packet in the flow, so that deviation from this number can be easily presented and read by the user. We have decided to neglect measurements above 1000 ms to avoid pauses in transmission and outliers that do not represent the overall state of the network. This timeout is based on the default RTO in TCP [41].

Variable	Description
flow_direction	PKT's flow in a TCP connection. The "direction" is arbitrary and does not imply whether it is incoming or outgoing. It stores information such as the flow's previously seen ACK, TSecr and pairing hashmaps.
rev_direction	The flow in the opposite direction of a flow direction's TCP connection.

Table 3.3: Explanation of the variables used in the pseudocode algorithms.

TCP SEQ/ACK number based pairing

Our tool implements finding packet pairs based on SEQ and ACK numbers as the primary method of inferring delay. In TCP, packets are sent with a SEQ number. This number, similar to timestamps, is echoed back to the sender in the ACK field, with one difference; the segment length in bytes of the sent packet is added to the ACK. Therefore, the program can create a variable that stores the SEQ number of an outgoing packet and adds its segment length to it. This results in a number it can pair with the ACK number of a subsequent packet in the opposite direction of the flow. The pseudocode for this operation can be seen in Algorithm 1, with an explanation of the variables used in the pseudocode in Table 3.3. This method can only be used to measure the RTT between outgoing packets carrying a payload and their companion ACK packets. We add regular outgoing data packets to the hashmap that indexes based on SEQ/ACK numbers, and retransmitted outgoing data packets to the hashmap that indexes based on timestamps. This decision is elaborated upon below.

Algorithm 1 Adding an outgoing TCP packet PKT to the Timestamp or SEQ/ACK pairing hashmaps.

```
if PKT is outgoing and PKT.seg_len > 0 then
  if PKT is a retransmission and PKT has timestamps enabled and
  PKT.TSval is not in flow_direction.ts_hashmap then
    flow_direction.ts_hashmap.add(PKT.TSval, PKT)
  else
    next_seq ← PKT.ACK + PKT.seg_len
    flow_direction.pair_hashmap.add(next_seq, PKT)
  end if
end if
```

It should be emphasized that retransmissions of incoming ACKs are not sampled. The reason for this is quite obvious; retransmitted ACKs are not indicative of the RTT of the connection.

There are a few factors to consider with pairing packets based on

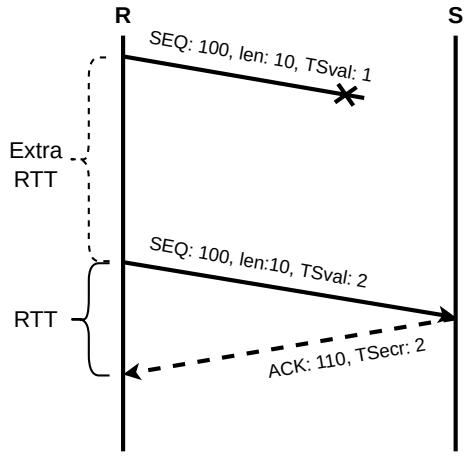


Figure 3.1: Retransmission RTT deviation. Pairing packets based on timestamps will not include the additional delay between the original transmission and retransmission of a packet (unless the time is so short that the retransmission happened in the same millisecond).

SEQ and ACK numbers. One such factor is that if an outgoing packet is lost and then retransmitted, it will still carry the same SEQ number. In contrast, the timestamp will be updated if the retransmission happens at least one millisecond after the original transmission. Without timestamps, TCP will not be able to differentiate between the original packet and the retransmitted one, meaning it can considerably over- or underestimate the RTT (see Figure 3.1). This can be solved by utilizing the TCP timestamp header option, elaborated upon further below.

For retransmitted outgoing data packets that do not enable TCP timestamps, the problem can be somewhat solved by filtering out the lost packets, and only counting the last transmitted data packet of the relevant SEQ number. Even if another packet, this time without data, is transmitted with the same SEQ number (which would otherwise result in a match for our packet pairing algorithm), it will not count it since it did not carry a payload. See Figure 3.2 for a visualization of this solution. The problem with this method is that there is no way of knowing for certain that the paired packets actually belong together (*i.e.,* if the first packet in Figure 3.2 was indeed lost), which can only be done with timestamps. Only the first instance of an incoming ACK number is counted, to give as accurate an RTT estimation as possible. It is important to note that this approach works for latency estimation, but for TCP, when calculating RTO, it does not. Disregarding packet loss would inevitably lead to a short RTO unrepresentative of the underlying network, and subsequent congestion caused by excessive retransmissions.

Packets using SEQ/ACK pairing will not include the added latency of a delayed ACK, as seen in Figure 3.3. The exception to this is if the delayed ACK is due to a server-side timeout when the extra delay is included. Therefore, RTTs of timestamp-enabled packets will more closely represent

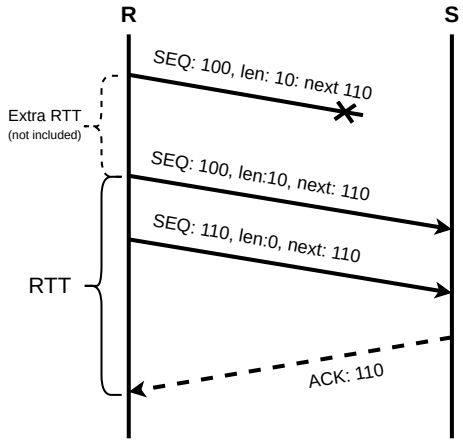


Figure 3.2: Solution to the retransmission RTT deviation problem for packets that do not enable TCP timestamps. The packet pairing algorithm calculates the next expected SEQ number only from the previously sent packet carrying data.

the delay that is relevant for TCP, whereas SEQ/ACK packet RTTs will more closely represent the fundamental network delay [48]. In this tool, the focus is primarily on representing the delay on the link, which is why SEQ/ACK pairing is our primary method of inferring delay. Sundberg et al. [48] concluded that TCP timestamps matching might result in an overestimated RTT because of delayed ACKs, but that SEQ/ACK matching would result in fewer RTT samples on lossy links. We solve this problem by primarily using SEQ/ACK numbers to pair packets, and TCP timestamps in the case of packet loss.

TCP Timestamp option based pairing

When a packet is sent, the current number of the TCP timestamp clock is set in the options header field, called the TSval (Timestamp value). The client which receives this packet then stores the TSval in its next outgoing packet's TSecr (Timestamp echo reply) field. When the original sender receives the new packet, it can look at the echoed timestamp value and compare it to its internal timestamp clock to get an estimation of the RTT, albeit one which does not take realistic network behavior into account [5]. The TSecr is only valid if the packet has the ACK flag set, which it is for most data packets. Similar to the SEQ/ACK packet pairing method, the RTT is only calculated for packets ACKing outgoing data packets for consistency and accuracy.

A 2022 study found that among the major operating systems, almost all had timestamps enabled by default [3]. It is also supported on most modern network devices and provides easily accessible and easy-to-parse measurement data.

Although the Timestamps option in the TCP header has many merits, there are a couple of problems with using timestamps to pair packets and

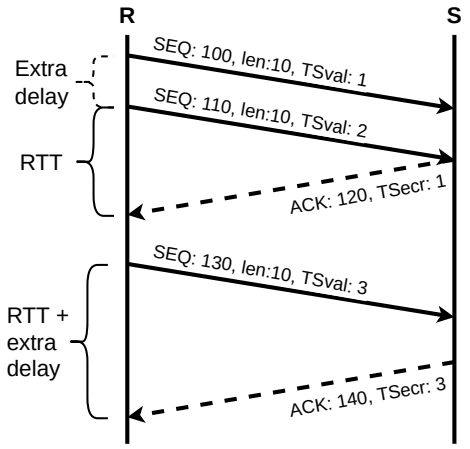


Figure 3.3: Delayed ACK RTT deviation. A packet ACKing the two first packets will match with the first one when pairing packets based on timestamps, and the second one otherwise. The delay is included either way if it is caused by a server-side timeout.

inferring delay. The last received packet's TSecr and its corresponding TSval packet might give the incorrect RTT because multiple packets may share TSecr values. This is solved by only counting RTTs from packets with updated TSecr values, ignoring any already seen values. In case multiple packets in the opposite direction share a TSval, only the first one is counted. Similar to our OWD measurements, unusually high RTTs (> 1 second) will be dismissed as they might be due to something like a longer pause in transmission. The pseudocode for the packet pairing algorithm for timestamp-enabled packets that match with a retransmission can be seen in Algorithm 2. The pseudocode for the packet pairing algorithm for non-retransmission outgoing packets can be seen in Algorithm 3. The result is a packet pair based on timestamps, visualized in Figure 3.4.

Algorithm 2 Finding the RTT of a packet pair when a new **incoming** TCP packet PKT is seen. Will only match with a timestamp-enabled outgoing retransmission of a data packet.

```
if PKT has timestamps enabled and PKT.TSecr ≠ flow_direction.TSecr then
    check_pkt ← rev_direction.ts_hashmap[PKT.TSecr]
end if
if PKT is ACK and check_pkt exists then
    RTT ← (Capture_time(PKT) - Capture_time(check_pkt)) ∗1000
    if RTT > 1000 then
        return Null {No RTT was found}
    end if
    return RTT
end if
```

TCP timestamps are not required to be enabled. Some servers may

Algorithm 3 Finding the RTT of a packet pair when a new **incoming** TCP packet PKT that is not a timestamp-enabled retransmission is seen.

```
check_pkt ← Null
if check_pkt was not found in TS hashmap and PKT.ACK ≠ flow_direction.ACK then
    check_pkt ← rev_direction.pair_hashmap[PKT.ACK]
end if
if PKT is ACK and check_pkt exists then
    RTT ← (Capture_time(PKT) - Capture_time(check_pkt)) ∗1000
    if RTT > 1000 then
        return Null {No RTT was found}
    end if
    return RTT
end if
```

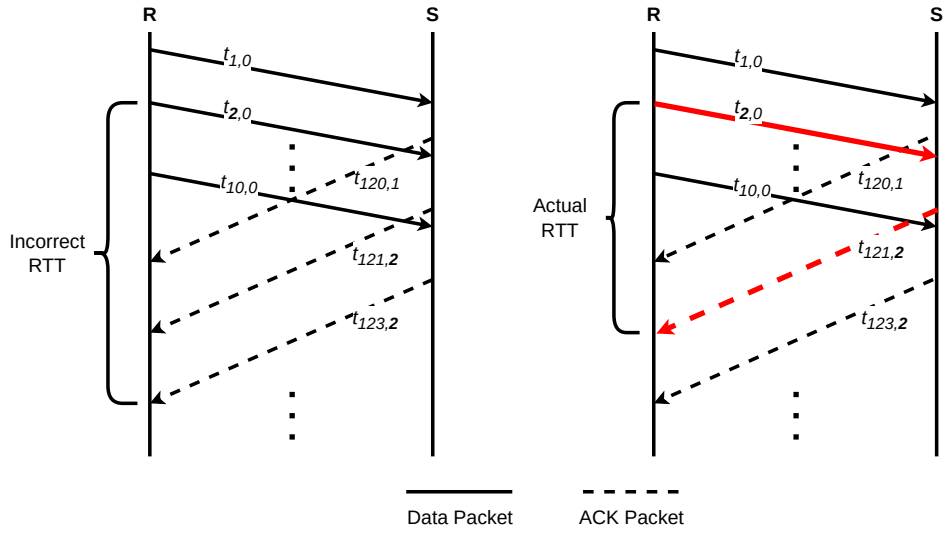


Figure 3.4: Finding packet pair RTT based on timestamps. Timestamps are written in the format: $t_{TSval,TSecr}$. R: Receiver, S: Sender.

disable them outright for security reasons, believing that the less access attackers have to data concerning the state of the network, the better. [3] also found that Windows, which is installed on a sizeable portion of home computers, fails to enable timestamps when a server requests it. Another problem with timestamps is that they are not very granular, measuring only in milliseconds. On a modern Internet connection, several packets can be sent in the span of a single millisecond, meaning it can be challenging to match a pair of packets sharing timestamp values perfectly. Using timestamps, it is possible to pair packets with corresponding timestamp values and then compare the capture times of the packets to calculate the RTT in microseconds.

Clock synchronization is not a substantial problem with this method, as either side has its own relative timestamp clock, and only the TSecr in incoming packets are considered in Algorithm 2. Equation 3.1 only considers the capture time of the paired packets measured on the same clock, *e.g.,* the one in the monitor. In certain physical conditions, clock skew might pose an issue. As previously mentioned, clock skew is not much of an issue in the case of this tool because of how little clocks usually skew, as well as how frequent the measurements it takes are.

Again, timestamp packet pairing will include the extra delay of a delayed ACK, whereas SEQ/ACK pairing will not. This is because the ACK number of the delayed ACK will match with the last sent packet instead of the original one, as shown in Figure 3.3. Only if a timeout on the server side causes the delayed ACK will the extra delay be included in the measured RTT of a timestamp-enabled packet. As a result, timestamps are only used for packet pairing when the outgoing data packets were retransmissions. In the case that timestamps are not enabled, packet pairs with retransmissions can not necessarily be trusted to belong together, as already mentioned.

Connection establishment and termination packets

In addition to packet pairing based on timestamps or sequence numbers, TCP traffic also allows us to find the RTT based on a connection's opening and closing packets. The RTT of these packets can be measured, as they will nearly always appear at the initiation and termination of a TCP connection, with only a few exceptions. This method works for both TCPs that enable timestamps and those that do not.

In the case of an *outgoing* connection opening, the time delta between capturing the SYN and SYN/ACK packets can be measured, as seen in Figure 3.5a. For the measurement to be valid, the SYN/ACK packet from the server being connected to must not be delayed or lost. The same goes for the FIN and FIN/ACK packets. Although the measurements are not always valid, a study did find that a similar technique yielded accurate results for most TCP connections [28].

In the case of an *incoming* connection opening, the time delta between capturing the SYN/ACK and ACK packets can be measured, as seen in Figure 3.5b. For the measurement to be valid, the ACK packet from the

server connecting to us must not be delayed or lost. The same goes for the FIN/ACK packet and its accompanying ACK packet. Whether a TCP connection is outgoing or incoming is determined by which side sent the original SYN packet. All packets with either of the SYN or FIN flags enabled are added to the pairing hashmap by SEQ/ACK pairing, as seen in Algorithm 4. The algorithm used determine which side opened the TCP connection and find the RTT between the establishment packet is in Algorithm 5.

Algorithm 4 Adding any packet PKT with the SYN or FIN flags enabled to the pairing hashmap.

```
PKT.next_seq ← PKT.ACK + PKT.seg_len
if PKT has either the SYN or FIN flags enabled and PKT is outgoing then
  flow_direction.pair_hashmap.add(PKT.next_seq, PKT)
end if
```

Algorithm 5 Measuring RTT using connection establishment packets when a packet PKT is seen, and SYN RTT is not already marked as found

```
check_pkt ← rev_direction.pair_hashmap[PKT.ACK]
if PKT is ACK and check_pkt exists and check_pkt is SYN then
  if PKT is retransmission or check_pkt is retransmission then
    mark SYN RTT as found {Mark as found, but disregard the result}
    return Null {No RTT was found}
  end if
  mark SYN RTT as found
  RTT ← Capture_time(PKT) - Capture_time(check_pkt)
  return RTT
end if
```

The algorithm works by looking at the packet where the previously seen packet in the opposite direction of the flow had the SYN flag set, so it is either the SYN or the SYN/ACK. It then checks if the checked packet is from a local device since it only measures the RTT between outgoing packets and responding incoming packets. If the checked SYN packet was outgoing, its capture time can be compared to that of the currently processed packet. When the SYN RTT is found, it is marked as such, and the program does not need to check the following packets for SYN RTTs. All packets with the SYN or FIN flags are added to the hashmap of previously seen packets in their respective flows. NETHINT will disregard the RTTs from the handshake if either of the relevant opening packets is a retransmission.

For connection termination packets, finding out which side sent the original FIN is vital. As either side can decide to terminate the connection, the FIN initiator cannot be correlated by whether the TCP is outgoing or incoming. A connection might also have begun before the monitoring tool started, meaning the monitor would be unable to tell who opened

the connection. Thus, an additional check looking at which side sent the first FIN packet is necessary. If the FIN was initiated from a local device, it will be the first to show up in the packet capture, and its capture time is compared to that of the responding FIN/ACK packet, as shown in Figure 3.6a. If the FIN was remotely initiated, FIN/ACK needs to be compared to the resulting ACK packet that marks the end of the connection, as shown in Figure 3.6b. This is achieved similarly to the way SYN packets are processed, by looking at the previous packet in the opposite direction. If that packet has the FIN flag enabled and is from a local device, the time delta in capture times between the two packets is marked; see Algorithm 6.

Algorithm 6 Measuring RTT using connection termination packets when a packet PKT is seen, and FIN RTT is not already marked as found

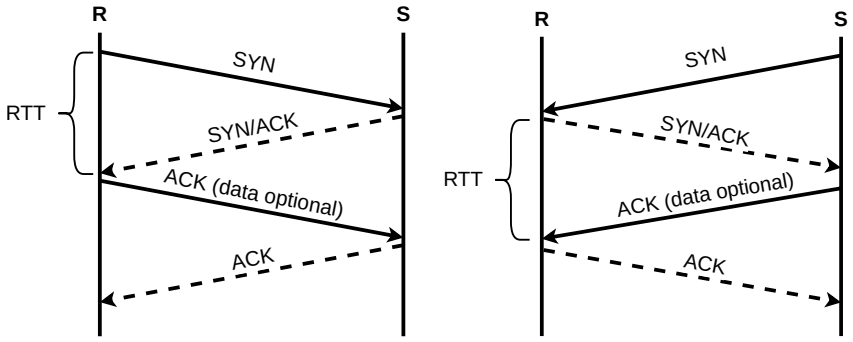
```
check_pkt ← rev_direction.pair_hashmap[PKT.ACK]
if PKT is ACK and check_pkt exists and check_pkt is FIN then
    if PKT is retransmission or check_pkt is retransmission then
        mark FIN RTT as found
        return Null {No RTT was found}
    end if
    mark FIN RTT as found
    RTT ← Capture_time(PKT) - Capture_time(check_pkt)
    return RTT
end if
```

This whole approach is limited in the sense that SYN and FIN packets only occur at the very beginning and end of connections. Nevertheless, as previously discussed, web transfers nowadays are usually short and frequent, resulting in a high number of packets for us to compare. Another shortcoming is the fact that connections may abruptly terminate, meaning the FIN packets are never exchanged.

TCP Fast Open (TFO) is a proposed way to send data along with the initial SYN packet of a connection, including a cookie in order to identify the user and remove the need for the full three-way handshake, similar to QUIC's 0-RTT connection resumption. Supporting TFO in our tool would mean a higher degree of both complexity and inaccuracy, as the initial SYN packet containing data might lead to processing time on the server, giving us a flawed estimate of the RTT. Although it is supported in MacOS and Linux, it is still not supported in most browsers. It will likely not be so in the future either due to protocol ossification and privacy concerns around the user-identifying cookie [40]. For these reasons, this tool will not implement support for TFO.

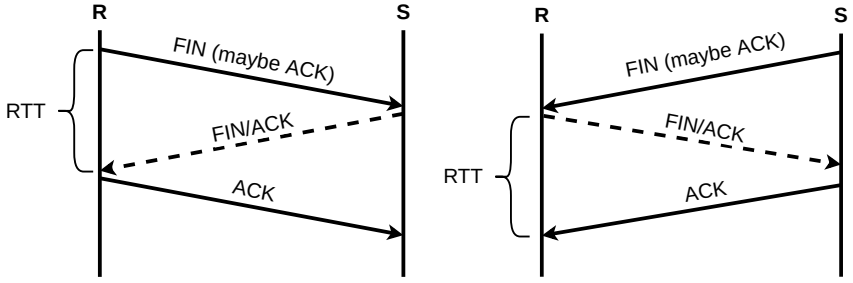
QUIC/UDP RTT

As already mentioned in Section 2.1.3, QUIC connections have the "spin bit", which, when enabled, allows an observer anywhere along the connection to infer the RTT. However, it is not required, and must be



(a) Outgoing TCP (b) Incoming TCP

Figure 3.5: Measuring RTT of SYN packets exchanged in the TCP three-way handshake. *R*: Receiver. *S*: Sender.



(a) Locally initiated FIN (b) Remotely initiated FIN

Figure 3.6: Measuring RTT of FIN packets exchanged in the TCP connection termination sequence. *R*: Receiver. *S*: Sender.

Delay source	Inputs
TCP SEQ/ACK based pairing	Pure ACKs
TCP Timestamps based pairing	Pure ACKs

Table 3.4: Unreliable delay sources of ***outgoing*** connections (not implemented).

supported by both ends of the connection. In testing on Chromium v109.0.5417.74, it was not enabled for Google.com or any other websites tested.

Instead, the RTT can be estimated by from looking at the capture time of the initial protection and handshake packets of a connection, as packet types are another exposed part of the header. QUIC connections are capable of either 1-RTT or 0-RTT connection establishments, so it is worth considering both scenarios. After these packets, though, it becomes impossible to differentiate between packets, and as such, impossible to measure delay.

Unreliable delay sources

The unreliable delay sources mentioned in Table 3.4 will not be implemented, as they cannot be trusted to yield accurate readings of the delay on the network. Contrary to outgoing data packets, pure ACKs (ACK packets without data) do not necessarily trigger the other side to immediately send an ACK packet in return, meaning the packet pairing algorithms for both timestamp- and non-timestamp-enabled packets would be inaccurate.

3.1.2 Inferring packet loss

Inferring packet loss is more straightforward than delay; however, as mentioned, it rarely occurs in real-world network situations compared to the ever-present packet delay. In TCP, the packet header can be analyzed and the present values compared to the TCP specification to discover information such as whether a previous segment was lost, or if the packet is a retransmission. A lot of checks are needed to confirm such events. However, in summary, a previous TCP segment is considered lost if its sequence number is higher than expected for the next incoming packet. A packet is considered a retransmission if, among a few other conditions, its sequence number is lower than expected, the segment length is 0, and it is not a SYN or FIN packet (the first and last packets of a connection, respectively).

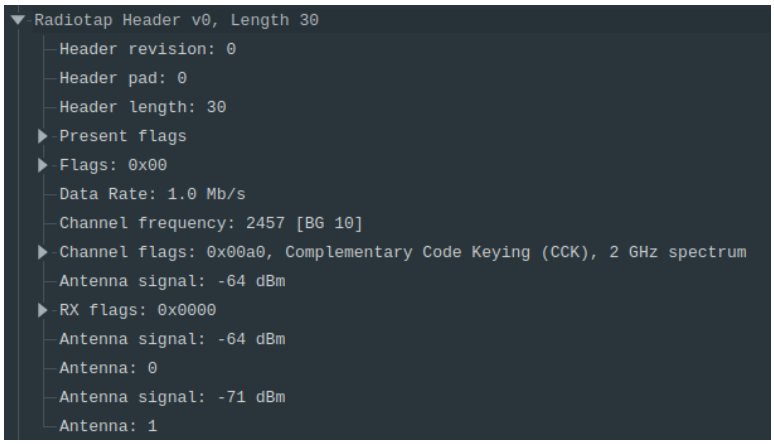


Figure 3.7: Radiotap header in a packet capture. Screenshot taken in Wireshark.

3.1.3 Measuring physical aspects from a transmitted packet

There are four different types of 802.11 frames: control, management, data, and extension. As mentioned in Section 2.5.1, setting the monitoring Wireless Network Interface Card (WNIC) to monitor mode means it is able to capture raw 802.11 packets, thereby capturing management and control packets as well as data packets. A way of gathering information about wireless packets is "Radiotap", which collects data from the Wi-Fi driver and sends it to userspace for processing. This is the standard packet format for sending and receiving 802.11 frames, and includes complementary data to the standard 802.11 WLAN radio information. It includes radio information such as the received channel frequency, antenna, and importantly, the antenna signal strength and noise for each receiving antenna. Figure 3.7 shows an example of the Radiotap header of a captured packet.

Finding the data rate of a transmitted packet is quite a complex operation and involves a number of steps to find. Data rates depend on the version of the 802.11 standard being used. Even finding this is complicated, involving a lot of header flag analysis before it can be inferred from the minimum supported properties. If the standard is found, the data rate still depends on various conditions, such as the number of spatial streams, modulation, coding, and channel width. These properties can then be indexed in what is known as an MCS (Modulation Coding Scheme) table, where each cell is a data rate. Due to the complexity of implementing data rate detection, it is considered outside of the scope of this thesis and, as such, will not be included. However, it is an interesting metric that has a lot of potential for future work.

3.1.4 Combining metrics for interference detection

Okada et al. [34] suggests measuring continuously increasing RTTs on the WLAN portion of the network over a period of one minute, and at the

end of the period calculating the total throughput of each connection. This was made possible by installing the monitoring point between the network switch and the AP and is therefore not an option for us. However, it raises an interesting point; if latency times increase over time, and in our case on several local devices once, it likely denotes a bottleneck on the WLAN. Connections that started with an especially low RTT are likely to go to a server that is relatively close to the receiver, meaning a smaller likelihood of interference on the relatively short network path. An increase in RTTs in these connections could then point to the likelihood of the locality of a bottleneck being in the WLAN of the receiver. The idea of a lot of the servers a user may access on a daily basis being located somewhat nearby is not as farfetched as it might once have been; service providers have increasingly built more and more data centers close to users in order to decrease latency, although studies have found this to be only a minor part of the solution of minimizing latency [26].

Another idea for correlating metrics to find the locality of a bottleneck is combining the various metrics that NETHINT supplies. For instance, if a flow's RTT increases at the same time as the signal strength for the transmitted data frames decreases, it is a good indication that the WLAN is limiting the connection. This is one of the reasons why a tool such as ePPing is not implemented. Even though it might run faster by hooking into the kernel, having only the RTTs available to the envisioned tool in user space is problematic as it is supposed to have a broad overview of everything going on at the WLAN. It might have been usable if the only metric collected by NETHINT was RTTs, but since it should have access to other data, such as signal quality and packet loss, it is not ideal for our use case.

3.2 The NETHINT tool

At its heart, NETHINT is a program that listens for packets and processes them according to their transport protocol. It is able to find reliable delay sources, detect lost segments and retransmissions, and present the data visually and save it to log files. For the purposes of this thesis, it is able to run in different "modes" depending on whether it is to process emulated traffic, traffic captured on the local WNIC of a host, or wireless traffic captured in monitor mode, as is its intended use.

The GUI is created to give a clear view of several metrics for the devices connected to the WLAN, which includes three main data plots with options to toggle each one on or off. All plots will link their X-axes, meaning that when the scale or view of one plot is manipulated, it is reflected in the other, making for easier viewing and analysis.

The way points are grouped and represented in the plots is based on running mode. By default, in wireless mode, the points are grouped by device. This means that the plot includes the RTTs for all flows to that device, and each device gets assigned a single color. When the tool runs in local or emulated mode, the points are grouped by flow instead, and each

one will have its own color.

The first plot is a scatter plot showing each valid RTT, grouped by device. In this plot, packet loss (packets classified either as retransmissions or where a segment has been lost) is shown as vertical lines in the same color as the group's points. Another of the plots is a scatter plot showing the RSSI (Received Signal Strength Indicator) for each of the points in the RTT plot. This plot is only enabled when the tool is running in wireless mode.

The final plot in the GUI is the relative OWD, displayed in a line chart. Each point shows the time delta between its OWD and that of the previous packet in the flow. This plot is somewhat unreadable on its own, but it incorporates an Exponentially Weighted Moving Average (EWMA) measure to show a more apparent development in values over time. The EWMA is calculated as:

$$EWMA_{pkt} = EWMA_{pkt-1} * (1 - \alpha) + OWD_{pkt} * \alpha \tag{3.2}$$

where $EWMA_{pkt-1}$ is the EWMA value of the previous packet in the flow, and

α is a value between 0 and 1 controlled by the user. It is represented in the GUI with a "smoothness"-slider that ranges from 0 to 100% (0% by default). The output of this slider (an integer between 0 and 100, x) is converted to an α value as:

$$\alpha = 0.83e^{-0.06x} \tag{3.3}$$

which makes the smoothness slider exponential.

3.2.1 How to use the monitoring tool

This subsection provides a brief guide on how to use the monitoring tool, including installation and invocation instructions, a list of available command-line arguments, and methods of exporting data from it.

Prerequisites

The tool requires at least Python v3.10, and the following libraries:

- Scapy v2.5.0
- NumPy v1.24.2
- Pyqtgraph v0.13.1
- PyQt6 v6.5.0

The following libraries are optional:

- mac_vendor_lookup v0.1.12

All required and opional libraries can be installed by running `pip install -r requirements.txt`.

Invoking the program

The tool is started by invoking the command `sudo -E python3 main.py`. Sudo privileges are required to run the program, as this is the only way Scapy can create a raw socket providing access to the underlying transport protocol. The `-E` flag is needed to preserve the environment when running the tool if the required libraries are installed on a non-root user, as recommended by the pip package installer.

The program takes arguments based on which mode it is being run in. All the available arguments can be seen by running `sudo -E python3 main.py -h`, the output of which is in Listing 3.1.

Listing 3.1: The output from running the main program with the -h flag.

```
usage: main.py [-h] [-ls] [-r [READ_PCAP]] [-w [WRITE_PCAP
    ↪ ]] [-s [SSID]] [-o] [-g] [-l [FILE]] [-t]
               [--wireless | --local | --emulated]
               [interface]

Process input arguments

positional arguments:
  interface             select interface by index

options:
  -h, --help            show this help message and exit
  -ls, --list-interfaces
                        list available network interfaces
  -r [READ_PCAP], --read-pcap [READ_PCAP]
                        read pcap file instead of from
                            ↪ interface
  -w [WRITE_PCAP], --write-pcap [WRITE_PCAP]
                        write pcap file
  -s [SSID], --ssid [SSID]
                        select SSID to listen to
  -o, --output          output packet capture to STDOUT
  -g, --gui             enable GUI
  -l [FILE], --log [FILE]
                        enable logging to [FILE]. Generates
                            ↪ filename if not specified.
  -t, --relative-time   display relative time in output(s)
  --wireless            (default) packets were captured in
      ↪ wireless monitor mode and have 802.11 headers
  --local               packets were captured on local NIC
  --emulated            packets were emulated and have
      ↪ fixed IPs
```

The tool takes only one positional argument, that being the index of the interface, which can be found by running the program with the `-ls` flag. It is also able to act as a tool to show live packet capture in STDOUT, enabled with the `-o` flag. When valid latency times are found, the tool will also write out information about them.

NETHINT has a visual GUI component, enabled with the `-g` flag. This will open a window with the plots as described above.

In order for the tool to only capture data packets from the WLAN that the user wishes to analyze, the `-s` flag can be passed with one or more arguments to define the SSID(s) of the relevant WLAN. The tool will then listen for 802.11 Beacon packets broadcasting the defined SSIDs and add the MAC address of the packets' senders to a list of "valid" MAC addresses to filter all data packets through. If the flag is not enabled, but the program runs in wireless mode, a wildcard (*) will be set as the target SSID, meaning all data packets from any WLAN in range will be passed through the filter.

The tool is not only capable of displaying a live capture, but can also read pre-recorded pcap and pcapng files. The `-r` flag can be supplied to enable this. It takes a single argument, the path to the pcap file to read.

The `-t` flag enables relative time, meaning the capture time of the first valid RTT will be set to 0, and all subsequent packets in the log and GUI will be calculated relative to it. The flags can be combined (*i.e.*, using `-got` instead of `-g -o -t`) for easier use. Some example uses can be seen in Listings 3.2 and 3.3.

Listing 3.2: Starting the program in wireless mode with interface wlan0mon, with GUI, output to STDOUT and logging enabled.

```
1 sudo -E python3 main.py --wireless -gol "log1.json" -i
    ↪ wlan0mon
```

Listing 3.3: Starting the program in emulated mode, with GUI enabled, reading from pcap file "emulated-test.pcap".

```
1 sudo -E python3 main.py --emulated -gr "emulated-test.pcap"
```

Export options

There are several ways of outputting data from NETHINT. Logging can be enabled with the `-l` flag. This will enable logging of all packets containing valid latency measurements, including details about the packet's connection, flow, capture time, signal if available, OWD, and RTT. The log will output to a JSON file, meaning both machines and humans can parse it relatively effortlessly. In order to specify the resulting log file's name, an argument can be passed to this flag. Otherwise, it will format the filename based on the current date and time.

The user can also export graphical plots by right-clicking in the GUI and choosing `export`. The user can choose the output format, either a CSV of the data, or an image or SVG vector file of the plot.

As well as reading them, the tool can write a new pcap file to a specified path with the `-w` flag and an argument. This can either be the result of live capturing packets on an interface or from another pcap file.

3.2.2 Monitoring tool implementation

NETHINT is written in Python, with its primary dependency being Scapy, which is used for sniffing packets. It relies on the Qt framework for the

backend graphics for the GUI. The GUI is mainly written using pyqtgraph, a Python library for plotting real-time data onto graphs. This makes the GUI both fast and extensible, which is ideal for this tool. The GUI is run in a separate process using Python's multiprocessing library so that it can add points to the plots without blocking the main packet dissecting process. It polls the queue of data points to be plotted every second and draws them on the graph.

The logging module works similarly when enabled, except that it runs in a separate thread instead of a process of its own. It is run whenever the thread gets scheduled and tries to get data points from the thread, which it subsequently writes to the log file.

Lifecycle of a packet

When a new packet is detected on an interface or read from the supplied pcap file, it is processed by a function that checks which protocol it uses. Based on the packet's protocol, it will be sent to a handler for either TCP or QUIC. 802.11 Beacon packets are also processed, but only by adding the MAC address of the source of the broadcast if the SSID matches one in the specified list of SSIDs to listen to.

If present, both the TCP and QUIC handlers also process the packet's RadioTap header. For all packets of types TCP, QUIC, or RadioTap, the packet is sent to a class constructor for the relevant class. NETHINT's custom packet classes are TCPPacket for TCP packets, QUICPacket for QUIC packets, and RadioTapPacket for packets with RadioTap headers. These classes parse the headers of the packets and find valid RTTs. Both TCPPacket and QUICPacket keep track of a packet's flow and connection. A *connection* is simply defined as two flows, one in the forward and one in the reverse direction. The direction of these flows has nothing to do with which side opened the connection. Instead, it is just a method of separating the flows. Each flow has a pointer to its reverse counterpart in order to compare the currently processed packet with the previous packets in the other direction. The flows also include data about the previous packet in the flow, a list of previous packets, and properties of the flow itself, such as whether the QUIC spin bit is enabled or not. These connections and flows each have a unique Connection ID (CID) and Flow ID (FID), which are referred to in the log output to make it easier to parse.

The TCP class is the most advanced, as there is a lot of potential data to analyze and correlate. Each packet goes through four essential steps:

1. Parsing
2. Analysis
3. Finding RTT
4. Updating its flow

In the parsing portion, information about the packets is parsed from the header, and the connection and flow of the packet are found and stored.

Properties such as the capture time and whether TCP timestamps are enabled are set at this stage.

During the analysis stage, the program uses the information gathered from the TCP header to infer additional details about the packet, such as whether the previous segment was not captured, is out of order, or is some kind of special packet like a zero window probe.

NETHINT then tries to find an RTT by using one of the reliable latency sources described in Section 3.1.1. It first checks for SYN or FIN packets to pair. If neither is found, it tries calculating the RTT by pairing either ACK numbers or timestamps. If timestamps are enabled, it will find the relative OWD also described in Section 3.1.1.

After the process of finding the RTT (and potentially relative OWD) is done, it moves on to updating the current flow with data such as the greatest seen SEQ number as well as timestamps and RTT. The packet is added to the list of packets in the flow and to the dictionary of packets to pair if it qualifies.

The process is similar for QUIC packets, except with fewer steps. The header is parsed, and the connection and flows are found before it moves on to finding the RTT. For QUIC, if the spin bit is disabled, only initial packets are checked. It is a relatively short process of calculating the time difference between the current initial packet and the previous one in the reverse flow direction. The flow is then updated, and initial packets are added to the list of packets in the flow. In the case that the spin bit is enabled, all packets are enabled and used to find the RTT.

Scapy implementation

The complete source code for NETHINT can be found in its GitHub repository [4]. Writing the tool using the Scapy library [42] opens up a lot of possibilities simply because it makes packet dissection so easy.

As the tool needs to be able to filter packets from one or more SSIDs, it listens for beacon packets broadcasting either of the target SSIDs. An example of how easily such a filter can be applied can be seen in Listing 3.4.

Listing 3.4: Finding MAC address of AP from beacon frames broadcasting the target SSID.

```
1 if pkt.haslayer(Dot11Beacon):
2     if pkt.info.decode('utf-8') == get_ssid() and not
      ↪ is_valid_ssid(pkt.addr2):
3         add_mac(pkt.addr2)
```

Since Scapy has no inherent support for QUIC packet dissection, we had to write a custom extension that defines some of the header fields. As QUIC has two header forms (long and short), the extension had to accommodate both cases. As mentioned earlier, QUIC is built on top of UDP, and as such, NETHINT was made to interpret all UDP traffic over port 443 as QUIC packets. This was performed with the built-in Scapy functions `bind_layers(UDP, QUIC, sport=443)` and `bind_layers(UDP, QUIC, dport=443)` to catch both outgoing and

incoming packets.

Scapy's automatic protocol interpretation makes it simple to obtain values. It extracts header information and makes it easily procurable via simple Python syntax, as can be seen in Listing 3.5.

Listing 3.5: Simple header value extraction using Scapy's protocol interpretation.

```
FIN = 0x01
SYN = 0x02

self.cap_time = float(pkt.time) # in us since epoch
self.rtt = None
self.owd_diff = None

# Parse IP header
self.ip_src = pkt[ip_ver].src
self.ip_dst = pkt[ip_ver].dst
self.port_a = pkt[TCP].sport
self.port_b = pkt[TCP].dport

# Parse TCP header
self.seq      = pkt[TCP].seq
self.ack      = pkt[TCP].ack
self.window   = pkt[TCP].window
self.flags    = pkt[TCP].flags.value
self.options  = pkt[TCP].options
self.seg_len  = len(pkt[TCP].payload)
self.next_seq = self.seq + self.seg_len
if self.flags & (SYN | FIN) > 0: self.next_seq += 1
```

This simplicity makes it straightforward to implement the algorithms seen in the pseudocode in Algorithm 1. The actual implementation of the packet pairing algorithm can be seen in Listing 3.6. Listing 3.7 shows how adding outgoing data packets to the hashmaps is implemented in the code.

Listing 3.6: Packet pairing algorithm in the source code.

```
check_pkt = None
if self.ts and self.tsecr != self.flow_direction.tsecr:
    check_pkt = self.flow_direction.rev.ts_pair.get(self.
        ↪ tsecr, None)

if not check_pkt and self.ack != self.flow_direction.
    ↪ last_ack:
    check_pkt = self.flow_direction.rev.pair_pkts.get(self.
        ↪ ack, None)

if (self.flags & ACK) > 0 and check_pkt:

    rtt = (self.cap_time - check_pkt.cap_time) * 1000
    if rtt > 1000:
        return None
    if get_output():
        print(f"Found valid RTT of {self.ip_src}>{self.
            ↪ ip_dst}: {rtt} ms.")

    return rtt
```

Listing 3.7: Packet pairing algorithm in the source code.

```
# SYN, SYN/ACK, FIN and FIN/ACK packets are always added
if (self.flags & (SYN|FIN)) > 0 and check_from_local(self):
    self.flow_direction.pair_pkts[self.next_seq] = self

# Only add data packets from local
elif check_from_local(self) and self.seg_len > 0:
    # Retransmission with timestamps enabled
    if (self.is_retransmission
            and self.ts
            and self.tsval not in self.flow_direction.
                ↪ ts_pair):
        self.flow_direction.ts_pair[self.tsval] = self

    else:
        # Add by next_seq for SEQ/ACK pairing
        self.flow_direction.pair_pkts[self.next_seq] = self
```

3.3 Monitoring hardware

The program is made to run on a Raspberry Pi 4 device, a small, low-power, relatively cheap single-board computer. The goal of running the measurement tool on this device is that it is easy to set up and replicate on other devices of its kind, potentially allowing for the distribution of SD cards with the monitoring software pre-installed to make it as simple as possible for the user to start monitoring and looking at results. The Raspberry Pi device is connected by an Ethernet cable to an access point to enable data transfer to and from it, as well as SSH or VNC (Virtual Network Computing) connections.

As previously mentioned, the WNIC needs to run in monitor mode

in order to receive wireless traffic going to all devices on the WLAN. However, the recommended OS for the Raspberry Pi 4, Raspberry Pi OS, does not allow for this setting to be enabled. Luckily, several solutions exist, such as compiling a custom version of the Linux kernel called the "Re4son-Kernel" with the setting enabled or installing a set of firmware patches called "Nexmon", which unlocks some extra functionality for the Broadcom chip that is used for Wi-Fi connectivity. Those options aside, installing the official ARM architecture image of the distribution Kali Linux proves to be the most straightforward option, as it comes pre-configured with Nexmon.

The built-in WNIC is still not ideal, as monitor mode is not officially supported and would change modes or stop working during our testing. It is, therefore, preferable to connect an external WNIC over USB in order to listen to the network and capture packets. The interface used for this thesis is the Alfa AWUS036ACM, which is one of the ones recommended by the Aircrack-ng community [2]. This is the community that made the Airmon-ng software, which is a tool for quickly and easily enabling monitor mode on a WNIC that supports it. We utilize said tool in this thesis.

3.3.1 Available information

Even with the listed hardware working as intended, a lot of data may still be unavailable to us. Packets can be encrypted at several layers, making it difficult for us as observers to get a lot of information from them.

Encrypted frames

The nature of the information that can be collected with the aforementioned WNIC varies on the wireless network's security configuration. Wi-Fi's first generation security algorithm, WEP (Wired Equivalent Privacy) [24] proved to be less private than the name implied, and was later replaced by other, more secure protocols, such as Wi-Fi Protected Access (WPA). As previously mentioned, putting the WNIC in monitor mode allows it to see control, management, and data packets for all Wi-Fi devices within range on the same channel. However, as most WLANs are encrypted with WPA 2 or 3, the data packets are all encrypted by the AP.

It is possible to decrypt WPA2-encrypted packets immediately after capturing them using their Pairwise Master Key (PMK), in this case, a Pre-Shared Key (PSK), between an AP and a receiving host. This requires capturing the initial authentication handshake between the two, packaged in an Extensible Authentication Protocol over LAN (EAPoL) packet. In WPA3, decryption is impossible without first obtaining the PMK that is issued during its authentication phase using Simultaneous Authentication of Equals (SAE). This is due to the fact that SAE exchanges are required to use Protected Management Frames (PMFs), which are protected from eavesdropping [54].

These security protocols are still susceptible to attacks such as the *deauth* attack, which sends a fabricated deauthentication packet to the AP, forcing

the targeted device to disconnect from it, breaking the connection, and disrupting service. An attacker would be able to force a client to reconnect using the *deauth* attack, whereby they could capture the EAPoL packet of the client when it reconnects to the WLAN. Implementing this into NETHINT would cross some ethical boundaries, which is not the intended use of this tool, and goes against our "silent listener" approach.

By waiting for EAPoL packets to show up on their own, NETHINT could implement some tools for on-the-fly 802.11 decryption, although Scapy has no inherent function for doing this. Instead, external tools could be utilized to achieve the same result. Dot11Decrypt is a somewhat popular C++ program that decrypts WPA2 by listening for EAPoL packets, obtaining the PSK for each connected device, creating a "tap" from a WNIC, where the decrypted packets can be read as from a regular interface [13]. Another potential solution is pyDot11, a Scapy extension to enable 802.11 packet decryption [47]. In the implementation process, issues were encountered while attempting to get either to work. As a result, rather than attempting to decrypt WPA2 packets and ignoring WPA3 (an AP can be configured to accept either type of encryption), this thesis looks at an unencrypted WLAN for simplicity's sake, but that is not to say it is impossible to implement in the future.

Encrypted headers

Although data packets sent on the Internet usually use either TCP or UDP for transport, an increasing amount of traffic utilizes the QUIC protocol to transport data. As mentioned in subsection 2.1.3, the transport layer protocol QUIC encrypts both its payload and most of its header, excluding some flags and a connection ID. It is, therefore, infeasible, if not impossible, to perform loss-correlation bottleneck detection on a QUIC connection.

Delay-correlation bottleneck detection, though, to some degree still possible to infer from looking at the incoming and outgoing packets of a QUIC connection, as the IP addresses and port numbers are still available to see on the underlying UDP header. Additionally, as mentioned in Section 2.1.3, there is an unencrypted flag in the QUIC header called the "spin bit", which is discussed in Section 2.1.3 and Section 3.1.1.

Chapter 4

Measurements

In this chapter, we will examine the various measurements performed in this thesis. We will start by detailing the emulation, including the emulated topology, how links are limited to confirm expected behavior, and what software is utilized to achieve it.

Next, we will describe the emulated process, scenarios and tests of the different emulated bottleneck situations. We will look at the results from the emulated tests and assess the accuracy of the tests in accordance with our expectations.

Finally, we will run a test with real devices on an open WLAN and see how NETHINT is able to gather and display metrics from two devices' wireless connections to a server on the same WLAN.

4.1 Emulation setup

The first step is to emulate traffic in a virtual environment so that we can confirm the NETHINT tool is working as intended, and can later extrapolate the implementation to look at real traffic. The emulation allows us to precisely set the capacity, delay, and queue lengths (*i.e.*, buffer sizes) of the links between the senders and receivers. We will verify if the tool is operating as expected by checking if it accurately calculates and displays network statistics and events, such as delay times and packet loss occurrences. It is vital to control the emulated links' capacity, as the capacity can be gradually lowered until it causes some congestion. If the capacity is limited to such a degree, it will be clear from the fact that all connections experience a drop in throughput simultaneously that the congestion is not coming from beyond the limited link. The delay between the receivers and r_1 is set to 0, as the monitoring device is spatially close enough to the senders that the time delta between NETHINT sniffing the packet and the end host receiving it is negligible. A more in-depth description of how setting up the emulated topology was accomplished is detailed in Section 4.2.1. We ran tests on this setup to understand how different capacities impacted flows, and the results can be viewed in Section 4.2.

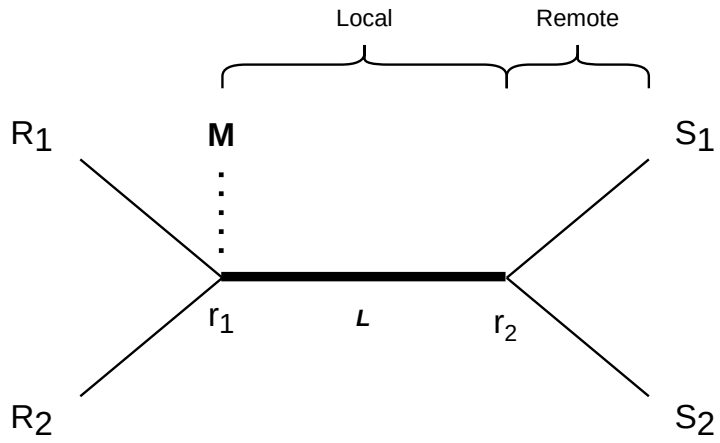


Figure 4.1: Emulation topology. *S*: Sender, *R*: Receiver, *r*: Router. The link *L* is considered the "local" portion of our network.

4.1.1 Emulation topology

The emulation topology consists of two senders and two receivers, connected by a common link *L*, as illustrated in Figure 4.1. In this case, the two receivers represent home devices, while the two senders represent web servers. As an example setup, *L* can be limited to 10 Mbps bandwidth, with 10 ms delay, representing the local wireless bandwidth. The link from r_2 to S_1 is limited to 1 Mbps, 20 ms delay, with a queue size limit of $2,500$ Bytes. The link from r_2 to S_2 is limited to 2 Mbps, 10 ms delay, with a queue size limit of $2,500$ Bytes. The reason for the differences in bandwidth and delay across the different links is that if they were configured with completely equal values, the connections would drop packets at close to the same rate as each other when their sending rate exceeded the bandwidth of their links. This would make the flows difficult to tell apart, a point which is illustrated in Figure 4.2.

BDP and queue sizes

Defining the queue size is crucial because the behavior of TCP depends on it to such a degree. The Bandwidth * Delay Product (BDP) is the product of the lowest bandwidth (capacity) of any path in a network measured in bits per second and the highest RTT measured in seconds. Its result is the maximum amount of data (in bytes) that can be in transit in a network link at a given time. By setting the queue size to equal the BDP, a TCP connection running for long enough should see the queue drain to 0 when it halves its sending rate after a congestion signal. This is because making the queue (buffer) the size of the BDP effectively doubles the capacity of the link, as the receiver can store as much data as the link can hold. In the case of our example, the queue sizes are calculated as

$$BDP = 1Mbit * 20ms = 125,000B * 0.02s = 2,500B \tag{4.1}$$

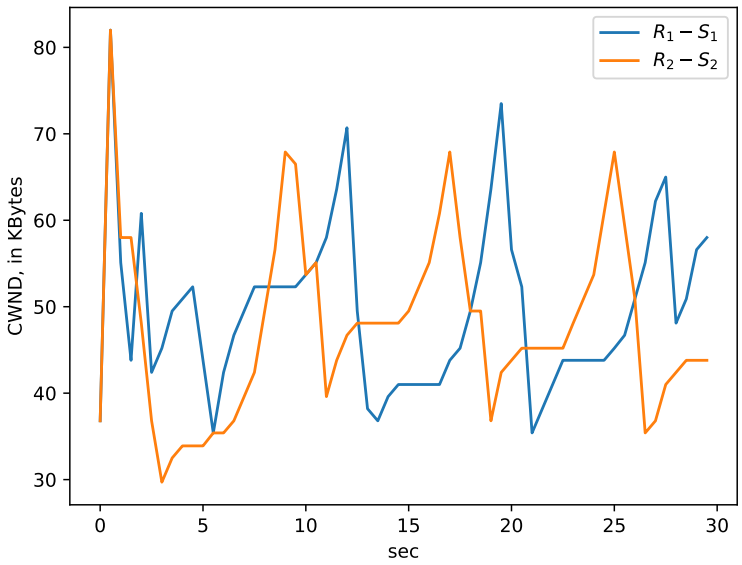


Figure 4.2: Two flows from senders with the same bandwidth and delay look almost identical in terms of packet loss. Parameters set here: RTT= 220 ms, capacity= 1 Mbps, queue size= 18 packets (BDP/1500 Bytes).

for r_1 - S_1, and

$$BDP = 2Mbit * 10ms = 250,000B * 0.01s = 2,500B \tag{4.2}$$

for r_1 - S_2.

Since ACK packets are so small, it makes sense to limit the queue size to a set number of bytes rather than a specific number of packets. This way, we can ensure that ACK packets do not artificially inflate and "block" the queue. It is also more reliable as TCP data packet segment lengths can vary, and even dividing the BDP by a typical packet size (such as 1500) often yields a decimal number, which tc is not able to accommodate for in a precise manner.

Bottleneck situations

In the case that the Wi-Fi network *is* in fact a common bottleneck for the hosts connected to the network, one host's drop in performance will be reflected in the other's. To test this case, the link from the two senders needs to be configured with a higher bandwidth than that of the shared Wi-Fi link *L*. If downstream packets now experience loss or delays simultaneously, it is because the link *L* limits it. We call this *Situation 1*.

In the case that the WLAN is *not* a common bottleneck, it would be wrong to assume that one flow's performance has any impact on another's since the part of the connection they share has such a high capacity. To test this case, the capacities of the links from the two senders must be limited. We call this *Situation 2*.

As the flows to senders with limited capacities would look very similar if they were configured with the same exact bandwidth and delay, they need to be limited with a slight difference. Consequently, when testing bottleneck situation 2, it will be split into *Situation 2a* for senders with equal limitations and *Situation 2b* for senders with unequal limitations. Using Mininet, a virtual network simulator, a simple network topology for testing and measuring RTTs can be created.

4.1.2 Emulation software

We will now provide an overview of the software that was used for traffic emulation. The tools used are Mininet for creating an emulated network topology, tc for simulating different network conditions, and iPerf3 for testing performance between each host on the emulated network.

Mininet

Mininet is a virtual network topology simulator designed to be realistic, running production code to simulate kernels, switches, and applications. The topology can be run entirely on a single machine, meaning it is relatively simple to set up and use. It allows the user to set parameters such as bandwidth, delay, packet loss, and queue size limit to emulate links of different performances. In our use case, it works perfectly to create a virtual network topology and run tests in order to gather data.

tc

tc (traffic control) is a program that is used to control the Linux kernel's packet scheduler. In fact, Mininet uses tc to implement its performance-limiting parameters over links. In this thesis, we use tc to configure bandwidth and delay times between the emulated links, as well as queue sizes on the servers.

The name tc refers to the management of queuing disciplines (qdiscs), which decide how a kernel handles incoming packets. The default qdisc in the Linux kernel is called pfifo_fast, and uses a somewhat extended FIFO queuing management mechanism. It divides the queue into three priorities, ensuring that packets of the highest priority class are always processed first [17]. The most basic qdiscs are pfifo and bfifo. Both limit the queue size and drop packets when they arrive at a full buffer. The difference is that pfifo limit the queue size by a set number of packets, whereas bfifo limits it by a set number of bytes. In our emulation, we use the bfifo qdisc, as the queue might be filled up by empty ACK packets, barely occupying any space yet saturating the queue.

iPerf3

iPerf3 is a network performance measurement tool written as a replacement for the outdated iPerf and iPerf2. It allows for tuning parameters

such as bandwidth, delay, loss, congestion mechanism, and more. iPerf3 is designed to find the maximum achievable TCP bandwidth on an IP network and is also capable of measuring UDP packet loss. It does this by checking the sequence number of the UDP datagram up against a counter and determining it as a packet loss if the sequence number is higher than 1 + the existing counter value [44]. UDP datagrams do not usually have sequence numbers, but iPerf3 encodes them into the first four bytes of the packet's data segment.

iPerf3 creates a client and a server, and emulates traffic by sending data from the former to the latter. By placing each node on a separate node in the Mininet virtual network topology, data packets can be transmitted and measured at a monitoring point between the two endpoints. This is helpful in analyzing how changes in the network path affect the traffic being sent over it.

4.2 Emulated tests

We can now combine the described tools to create a virtual network topology and start adjusting the links to see how traffic behaves when traversing the links.

4.2.1 The emulation process and configuration

With our Mininet setup, using the topology shown in Figure 4.3, an instance of tcpdump is run on the r_1 router, specifically on its interface connecting to r_2. As the link L represents the local portion of the network, the shape of the traffic going over it, *i.e.,* between r_1 and r_2, will be adjusted in order to emulate a bottleneck on the local network. Similarly, the traffic shape between r_2 and the senders S_1 and S_2 will be adjusted to emulate a bottleneck elsewhere in the network. Again, there is no set limitation on data between the emulated receivers representing local devices, R_1, R_2, and their router r_1. Mininet is only used to create the virtual nodes (hosts and routers) and the links between them. It allows us to set up a virtual network topology, name each interface, and set each interface's IP address, as shown in Listing 4.1.

Listing 4.1: Adding the routers and the link L between them.

```
1 r1 = self.addHost('r1', ip='10.1.0.1/24',
2                   defaultRoute='via 10.1.0.2', cls=
                        ↪ LinuxRouter)
3 r2 = self.addHost('r2', ip='10.1.0.2/24',
4                   defaultRoute='via 10.1.0.1', cls=
                        ↪ LinuxRouter)
5
6 self.addLink(r1, r2, intfName1='r1-r2', intfName2='r2-r1',
7              params1={'ip': '10.1.0.1/24'},
8              params2={'ip': '10.1.0.2/24'}, cls=TCLink)
```

The LinuxRouter class is simply a Mininet node which is issued the command

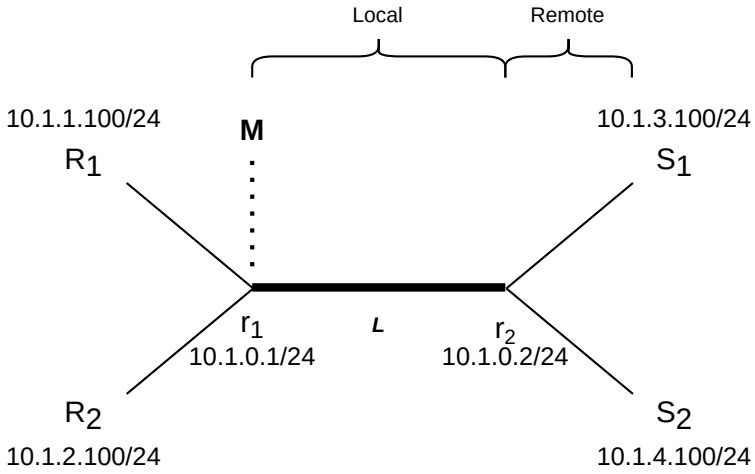


Figure 4.3: The emulated network topology, with IP addresses.

```
self.cmd("sysctl net.ipv4.ip_forward=1")
```

in order to enable routing.

Mininet does have built-in functionality allowing the user to change bandwidth, delay, queue lengths, and more by passing arguments to the `addLink()` function. In our testing, however, it seemed as if the queue was applied in both directions on the link, and was less reliable overall than desired. Luckily, Mininet allows the user to run commands directly on the emulated nodes in the network, meaning tc can be used to apply a bespoke queuing discipline of our own making. As such, the following commands were run to specify delay, capacity, and queue size in bytes on the interface called r1-r2, shown in Listing 4.2.

Listing 4.2: Configuring queuing discipline for r_1's outgoing interface to r_2 with delay=10ms, bandwidth=20Mbit.

```
1 r1.cmd("tc qdisc del dev r1-r2 root;\
2     tc qdisc add dev r1-r2 root handle 2: netem delay 10ms
      ↪ ;\
3     tc qdisc add dev r1-r2 parent 2: handle 3: htb default
      ↪ 10;\
4     tc class add dev r1-r2 parent 3: classid 10 htb rate 20
      ↪ Mbit;")
```

The same thing was done for the links between r_2 and S_1, S_2, shown in Listing 4.3.

Listing 4.3: Configuring queuing discipline for r_2's interfaces to the emulated senders.

```
r2.cmd("tc qdisc del dev r2-S1 root;\
    tc qdisc add dev r2-S1 root handle 2: netem delay 20ms
        ↪ ;\
    tc qdisc add dev r2-S1 parent 2: handle 3: htb default
        ↪ 10;\
    tc class add dev r2-S1 parent 3: classid 10 htb rate 10
        ↪ Mbit;\
    tc qdisc add dev r2-S1 parent 3:10 handle 11: bfifo
        ↪ limit 25000;")
r2.cmd("tc qdisc del dev r2-S2 root;\
    tc qdisc add dev r2-S2 root handle 2: netem delay 10ms
        ↪ ;\
    tc qdisc add dev r2-S2 parent 2: handle 3: htb default
        ↪ 10;\
    tc class add dev r2-S2 parent 3: classid 10 htb rate 20
        ↪ Mbit;\
    tc qdisc add dev r2-S2 parent 3:10 handle 11: bfifo
        ↪ limit 25000;")
```

This is where queue sizes are applied with the queuing discipline bfifo.

In some cases, interfaces may perform hardware offloading, such as TCP Segmentation Offload (TFO) and Generic Segmentation Offload (GSO). These offloading mechanisms utilize the NIC hardware to segment a frame into several smaller segments to reduce CPU overhead [29]. Though efficient in real-world use, they can make traffic act in unforeseen or unexpected ways and are therefore disabled using the `ethtool` command.

Listing 4.4: Disabling hardware offloading for the interfaces at the emulated hosts R_1 and R_2 to the router r_1.

```
R1.cmd("ethtool -k R1-r1 tso off;\
        ethtool -k R1-r1 gso off;\
        ethtool -k R1-r1 lro off;\
        ethtool -k R1-r1 gro off;\
        ethtool -k R1-r1 ufo off;")
R2.cmd("ethtool -k R2-r1 tso off;\
        ethtool -k R2-r1 gso off;\
        ethtool -k R2-r1 lro off;\
        ethtool -k R2-r1 gro off;\
        ethtool -k R2-r1 ufo off;")
```

After this, iPerf3 clients were run on the receivers, and servers were run on the senders. Since iPerf3 clients send data to the servers, this might at first appear to conflict with our earlier statement about how local end-user devices are usually the ones receiving data. However, since our monitoring tool mainly measures the delay between outgoing data packets and incoming packets ACKing them, sending data from the emulated senders to the receivers resulted in barely any data points to analyze.

There was also an attempt to run an nginx[1] server on the senders. The idea was that it would emulate real-life traffic more closely yet still yield

[1] nginx is a web server [32]. It was chosen for its simplicity and good performance.

reproducible and measurable data. However, the same issue arose as the only data packets going from the receivers were the HTTP GET requests, and the concept was ultimately dismissed due to these inconclusive results.

Since the interfaces were named, the command

```
r1.sendCmd("tcpdump -i r1-r2 -w results/result.pcap")
```

can be run to sniff emulated packets at r_1's interface named r1-r2, and write the result to a pcap file. The pcap files were then processed using our monitoring tool to extract the reliable delay times as well as getting a visual representation of them.

4.2.2 Testing scenarios

Testing started with some initial tests to confirm that the testing setup worked the way it was expected to, and to get some data to compare real-world tests to later.

Tests were then conducted according to the scenarios defined in Section 4.1.1, wherein Situation 1 means that the WLAN *is* a common bottleneck for flows, and Situation 2 means that it is *not*. For the emulation's sake, this means that depending on which situation is being tested, either the link L or the links between r_2 and the two senders (r_2-S_1 and r_2-S_2) need to be limited in some capacity. The former link represents the local segment of the path, while the latter two represent the remote segment.

This thesis will apply two different scenarios, illustrated in Figure 4.4. The first scenario is an all-virtual, emulated environment. The topology is elaborated upon below. This scenario is practical because no physical setup is needed to start performing tests. Every part of the topology can easily be altered and adjusted to fit the tests.

Then, a setup is created where the sender is on the local LAN, and the clients talk wirelessly to it through the AP.

In Figure 4.4, a proposed third scenario is also depicted, where two clients on the WLAN communicate with a sender that is located remotely. This test could help evaluate the efficacy of the monitoring tool in its intended use, and is discussed in more detail in Section 5.2.

4.2.3 Emulation results

Our monitoring tool's GUI allows us to sniff and process packets either as they appear in real-time, or from a pcap file. In the case of this emulation, it was found to be most effective to write to a pcap file first, then later analyze it in the monitoring tool, making it possible to store and keep the results for later comparison.

4.2.4 Initial tests

Before looking at how traffic behaves in the pre-defined situations from Section 4.1.1, some tests were run to see how simple configurations looked

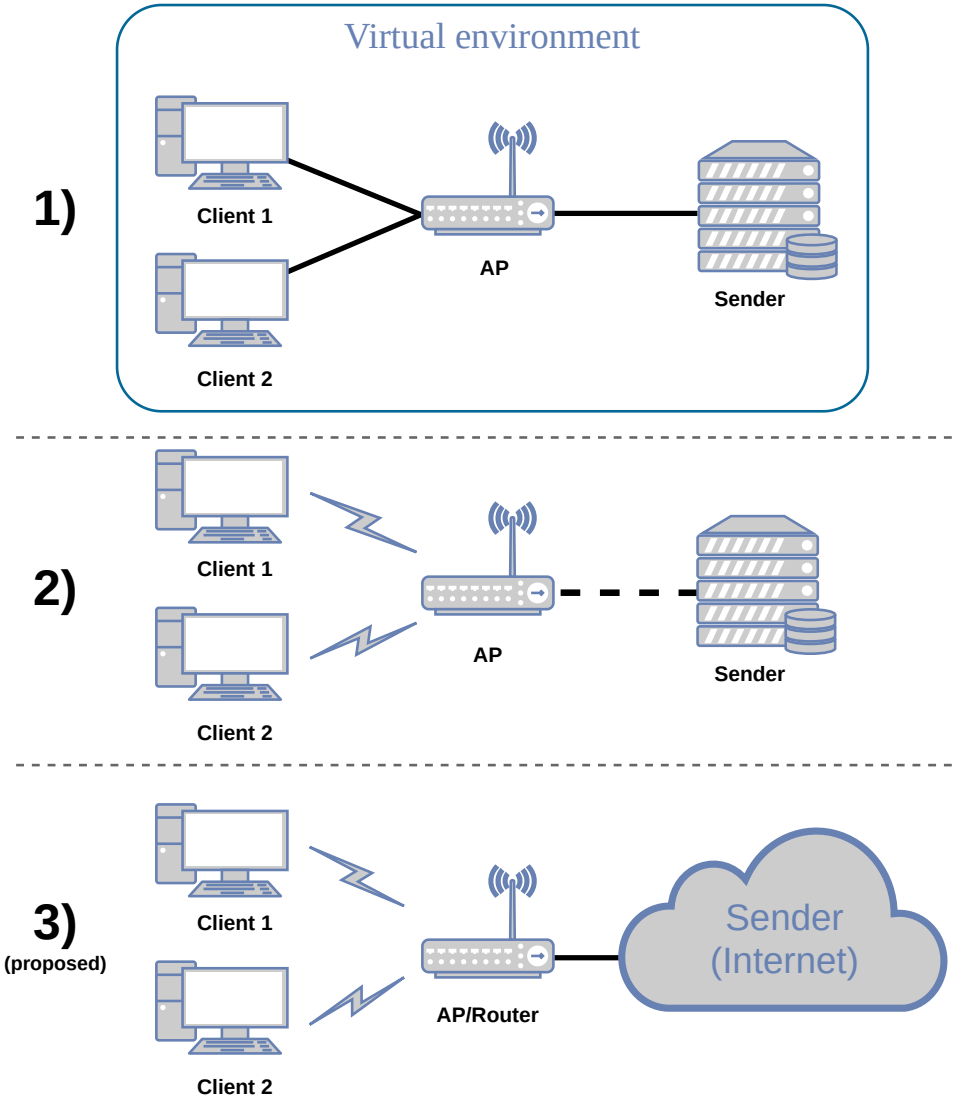


Figure 4.4: The two scenarios employed while testing the monitoring tool, as well as a proposed third one. 1: Emulated, 2: Local, 3: Remote.

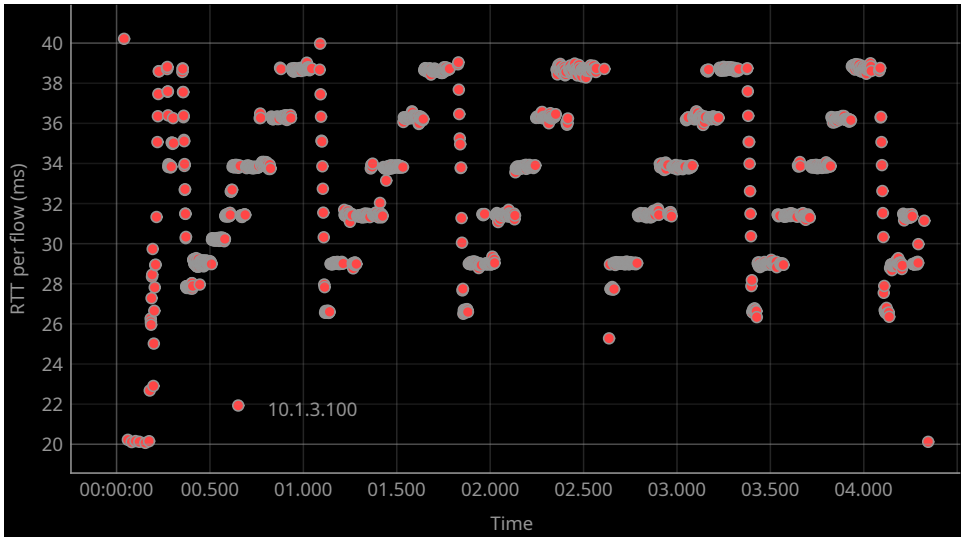


Figure 4.5: One flow with queue size=BDP, Capacity=10 Mbit, delay=20 ms.

when processed through the monitoring tool, and to test that the emulation setup worked as expected. One noticeable point is that when enabling latency-finding packet pairing on SYN packets, those packets always had an RTT of twice the base delay on the link, whereas the RTT from FIN packets was always equal to the delay on the link. Although not entirely certain, we theorize that this is due to a process in the emulation tools used, such as Mininet, since the same behavior is not seen in later testing over actual Wi-Fi.

Test 1: One flow (BDP)

With only one flow, there is no contention on the link, and as such, the delay times are expected to gradually increase over time as the buffer on the other end is filled up, then drop back down when packets are consumed. This flow is between R_1 and S_1. For the first test, the queue size was set to equal the BDP of the link. The exact parameters set on the remote link r_2-S_1 for Test 1 were:

- Capacity = 10 Mbit
- Delay = 20 ms
- Queue size = BDP = 25,000 Bytes

The emulated local link L was not limited for this test. The results were as expected, with the RTT starting at 20 ms and going up to almost exactly double that value, as can be seen in Figure 4.5.

Test 2: One flow (half BDP)

Configuring the queue size to be half the BDP and keeping the same bandwidth and delay parameters, the RTTs were expected to rise to around

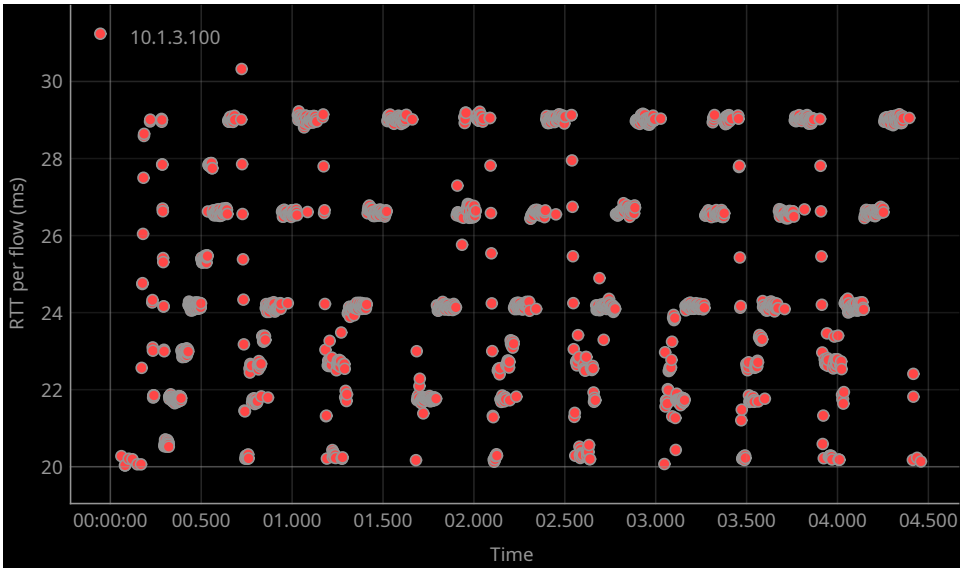


Figure 4.6: One flow with queue size=½ BDP, Capacity=10 Mbit, delay=20 ms.

1.5x that of the base RTT due to the fact that a halved queue means half the increase in RTT from the base. The exact parameters set on the remote link r_2-S_1 for Test 2 were:

- Capacity = 10 Mbit
- Delay = 20 ms
- Queue size = BDP = 12,500 Bytes

Again, the local link L was not limited in bandwidth or delay. Results for this test were again as expected, and the monitoring tool showed packets going between 20 and 30 ms, almost precisely 1.5x the base RTT of 20 ms. The results are visualized in Figure 4.6.

Test 3: Two flows (BDP)

For this test, the shared local link L is set to the sum of the two separate links, meaning there should be just about enough capacity for the two flows. The exact parameters set for Test 3 were:

Local portion (L)

- Capacity = 20 Mbit
- Delay = 0 ms

Remote portion (both r_2-S_1 and r_2-S_2)

- Capacity = 10 Mbit

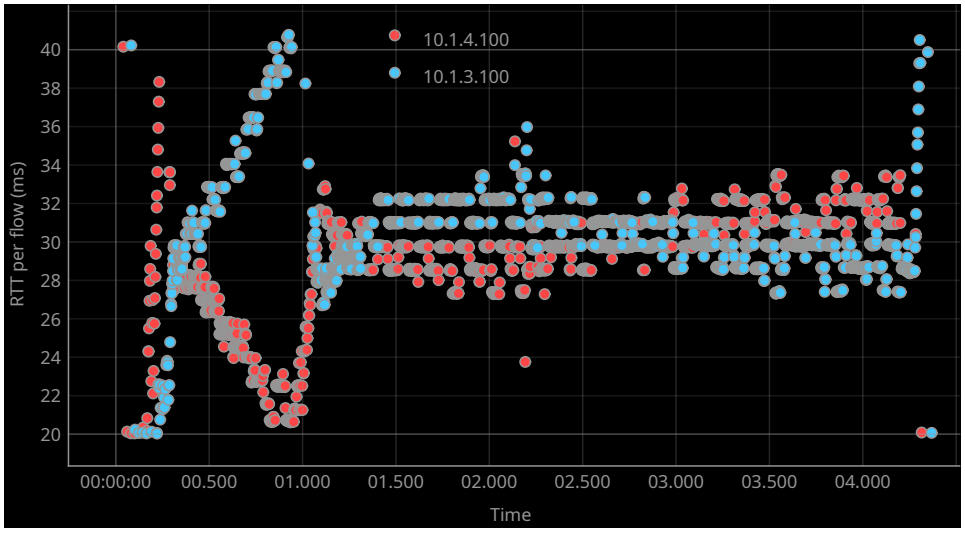


Figure 4.7: Two flows with the same limitations on both the local and remote link contend at first but then balance out their sending rates.

- Delay = 20 ms
- Queue size = BDP

Looking at the visual display of the result in Figure 4.7, the flows are at first competing. The first flow to get a congestion signal is then much slower to increase its sending rate, while the other flow is probing for capacity. Subsequently, when the second flow receives a congestion signal, it backs off. The two flows then converge to similar RTTs around 30 ms, almost exactly between the initial RTT of 20 ms and the highest expected RTT of 40 ms.

4.2.5 Situation 1

Now that it is known that the emulated setup and monitoring tool both behave predictably, tests can be run for the bottleneck situations. In this situation, the local portion of the network is the bottleneck. The exact parameters set on the local link L were:

- Capacity = 20 Mbit
- Delay = 20 ms
- Queue size = BDP

The remote portion of the network was not limited. The results of this test, which can be seen in Figure 4.8, show that the packets mostly follow the same pattern of RTTs, with a couple of outliers. This shows us that it is, to some degree, possible to infer a bottleneck on the WLAN in a given time period if several flows share the same delay characteristics in that time.

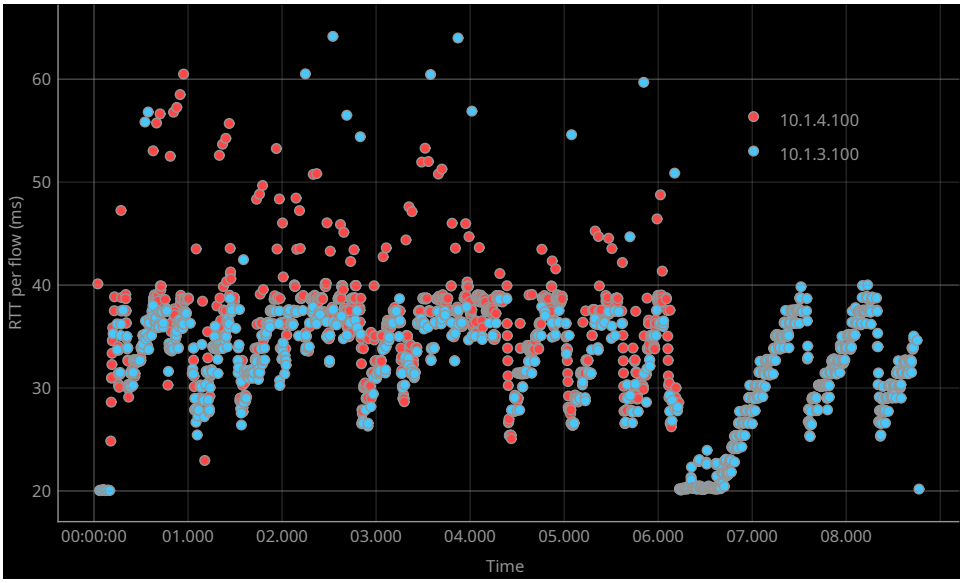


Figure 4.8: Situation 1: Two flows competing on a bottlenecked local link.

4.2.6 Situation 2a

Situation 2a is the situation where the bottleneck for the flows is not shared, but they have equal limitations on the remote links. The parameters set for both remote links r_1-S_1 and r_1-S_2 were:

- Capacity = 10 Mbit
- Delay = 20 ms
- Queue size = BDP

With two flows, both links being configured to be equal, and the local link *L* configured to have no delay or capacity limitations, there is no contention as both flows are limited by their own separate links. As expected, though, this is hard to tell since the flows look so much alike, as can be seen in Figure 4.9. Therefore, an adjustment of one of the limited flows is required.

4.2.7 Situation 2b

In Situation 2b, the bottleneck for the flows is still not shared, but this time, the two senders have different configurations on their remote links. We have done tests limiting both the capacity and the delay in turn, in order to see how each configuration impacts the shape of the traffic.

Adjusting capacity

The first parameter that was limited was capacity. This limitation should not have any impact on delay times, but should take significantly longer to finish, as less data can be sent per round-trip. The parameters set when adjusting the *capacity* on one flow were:

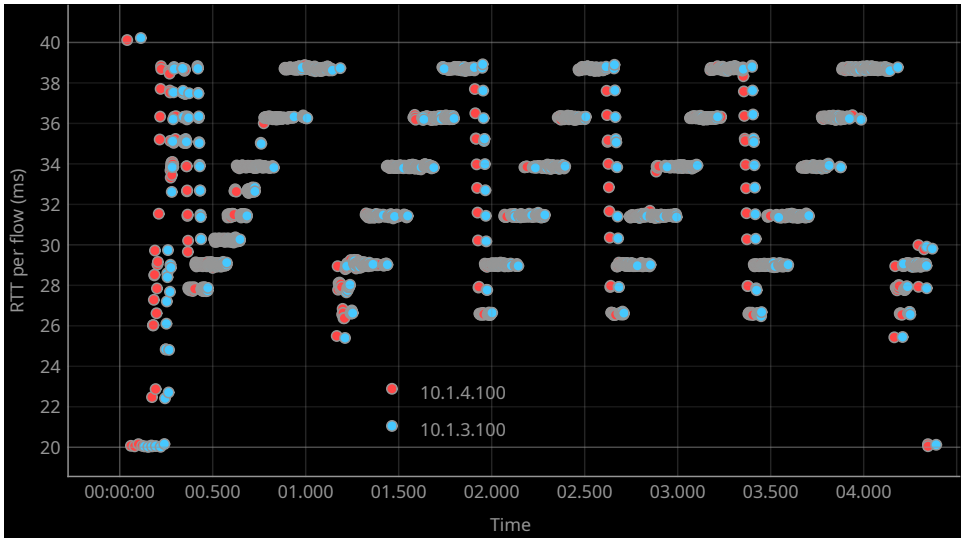


Figure 4.9: Situation 2a: Two flows, though without a shared bottleneck, will look almost indistinguishable when they are configured with the same characteristics.

Sender 1 - IP: 10.1.3.100

- Capacity = 10 Mbit
- Delay = 20 ms
- Queue size = BDP

Sender 2 - IP: 10.1.4.100

- Capacity = **5 Mbit (adjusted)**
- Delay = 20 ms
- Queue size = BDP

When the capacity is halved in one flow, the flows become distinguishable in that the one with the least capacity will, as expected, send significantly fewer packets per second and, as a result, has to continue sending for twice as long as the higher-capacity flow, as can be seen in Figure 4.10.

Adjusting delay

When limiting the delay on one flow, the expectation was for the adjusted flow to have a higher response time while not having to keep sending for longer than the other flow. The parameters set when adjusting the *delay* on one flow were:

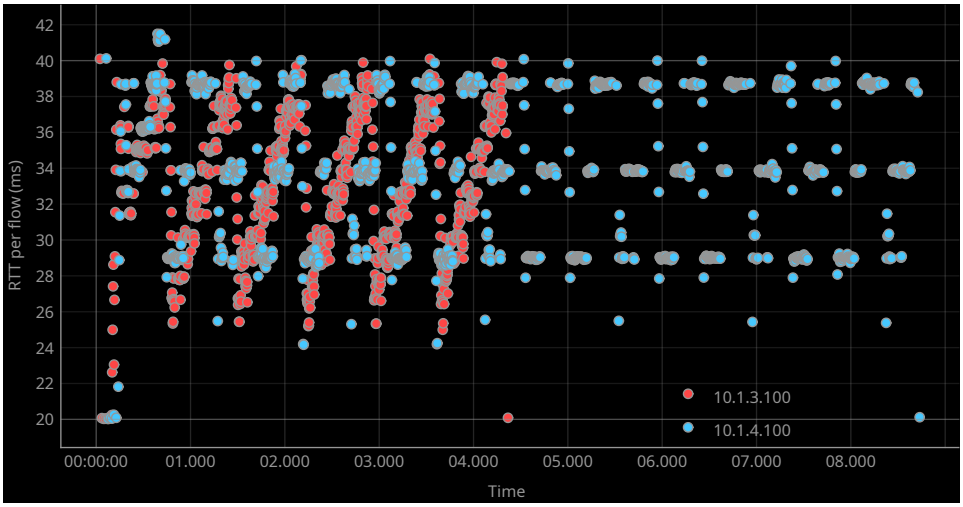


Figure 4.10: Situation 2b: Adjusting the capacity of one of the remote links.

Sender 1 - IP: 10.1.3.100

- Capacity = 10 Mbit
- Delay = **15 ms (adjusted)**
- Queue size = BDP

Sender 2 - IP: 10.1.4.100

- Capacity = 10 Mbit
- Delay = 20 ms
- Queue size = BDP

As can be seen in Figure 4.11, when the delay is limited on one flow, both flows finish sending at approximately the same time, even though the latency of each packet is half as much for one of the flows.

4.3 Local measurements

The physical setup for local measurements is an isolated WLAN with one AP, the channel set to one that does not interfere with surrounding existing WLANs. The only connected devices are the ones stated. The connected devices are:

- Raspberry Pi 4 Model B (Server)
- Raspberry Pi 3 Model B+ (Host 1)
- MacBook Pro 2015 (Host 2)

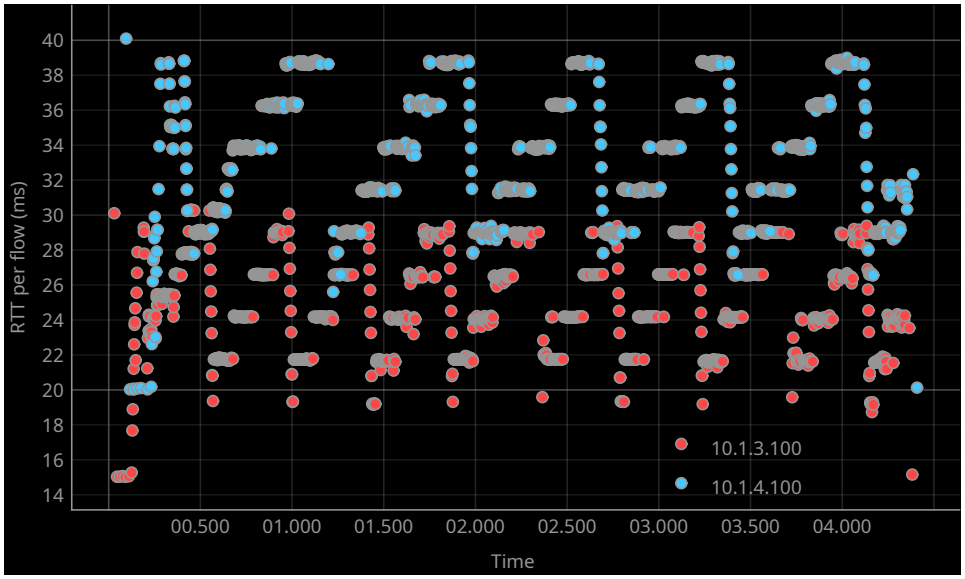


Figure 4.11: Situation 2b: Adjusting the delay of one of the remote links.

In this test, the two hosts sent data from their own respective iPerf3 clients to an iPerf3 server on the server. An interesting point with this test is that even though the hosts sent the same amount of data, and the flows finished at almost the same time, the MacBook sent significantly fewer packets in total than the Raspberry Pi. After inspecting the packet capture, it became apparent that this was due to the MacBook consistently sending large packets (1500 Bytes) The Raspberry Pi was more inconsistent, often sending packets around 130 Bytes. The result can be seen in Figure 4.12.

It is also of note that RTTs of the packets in the different flows seem to follow the same general shape. This could be due to the fact that they not only share the physical wireless medium, but also have a common server to which they are sending data.

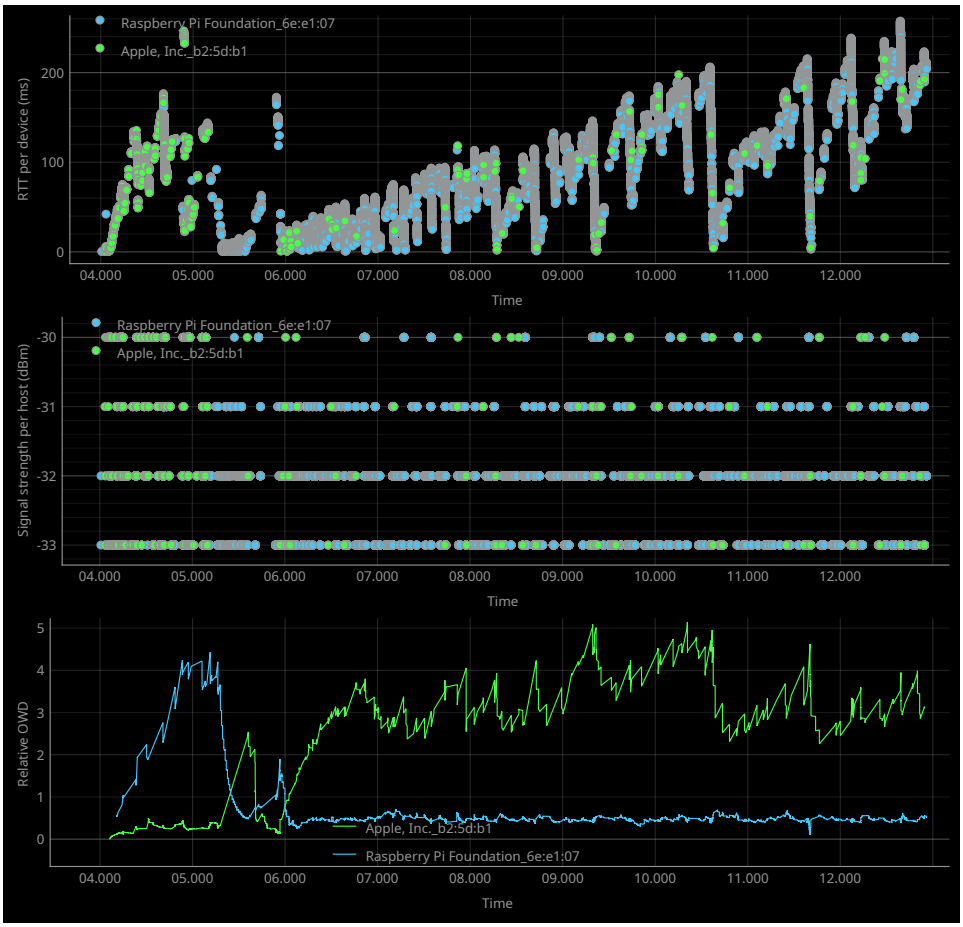


Figure 4.12: Local test: Two senders each sending 50MB to one server. The OWD "smoothness" is set to 100%, or EWMA $\alpha \approx 0.00758$.

Chapter 5

Conclusion and Future Work

In this thesis, we have presented the NETHINT tool, which finds statistical Wi-Fi metrics, laying the groundwork for correlating them and be able to see whether or not the bottleneck in a connection exists on the local network or at another point in the path. We have shown that it collects data correctly by having it analyze and display predictable TCP behavior. It is able to log metrics over time and let the user export the data in several different ways, opening up a lot of potential for future work and implementation.

5.1 NETHINT and the state of the art

This tool combines finding RTTs from reliable delay sources with a WLAN-wide monitor, displaying data in a user-friendly way that gives the user a clear overview with which to infer the state of their local network at a point in time. By monitoring several devices at once, there is much potential for comparing statistics about data streams where other tools only look at the device they are running on.

5.2 Future work

NETHINT, as presented, is a framework that has the potential to be built upon in order to make it into a fully-fledged network monitoring and interference-finding tool. For this to happen, we deem some capabilities crucial to implement, as well as some alternative features which could be worthwhile to work into the tool. There are also some additional tests and experiments that could help collect data in order to improve the tool.

5.2.1 Critical functional enhancements

We consider three main prospective components the most critical, those being 802.11 decryption, data rate measurement, and devising a WLAN interference metric.

As mentioned in Section 3.3.1, 802.11 decryption for WEP, WPA, and WPA2 is a solved problem, only needing implementation into either the tool or the underlying hardware. Methods of doing this are discussed

further in Section 3.3.1. Even though there is currently no way of decrypting WPA3-encrypted frames going to and from a device without obtaining the shared key, which is transmitted in a protected packet, many homes still use WPA2, making the functionality a very relevant candidate for addition to the tool.

Another welcome addition would be the measuring of wireless data rates. As mentioned in Section 2.3.4, the wireless physical sending rate is an important metric that can indicate a lot about the state of the WLAN, especially in networks that do not support an airtime fairness-based queue. Also mentioned in Section 2.3.4, one device's drop in data rate may severely impact the performance of other devices due to the 802.11 performance anomaly. By looking at the minimum supported protocols in the RadioTap header of a packet, it is possible to find the 802.11 standard in use, and use that information to look up which data rate was used for the packet.

The last crucial feature for NETHINT is the ability to formulate a kind of interference metric that can clearly tell the user of the tool whether or not their WLAN was experiencing issues due to local interference at a given point in time. Section 3.1.4 details some ideas around this topic. While it would be an option to calculate and display multiple metrics based on the available data, an easy-to-understand signal, like a traffic signal or "interference score" indicating the likelihood of a local bottleneck would be preferable.

5.2.2 Additional functionality

Although less crucial than the three main envisioned aspects of the tool, some features could prove to be beneficial to include.

One feature would be differentiating between uplink and downlink packets. Since the tool already measures both in parallel, it is feasable to separate them. This could be beneficial as queues can be caused in either direction and are not necessarily correlated, making for some interesting additional data points.

Another component could be measuring jitter, or the variation in latency on a packet flow. This could be done by measuring the average RTT in short intervals, then comparing the measurements over time. This could speak to the stability of a path in terms of congestion and latency, and might be yet another data point to compare between the devices on the WLAN. Another simple addition could be sending a probe request packet on start (with -s flag set, and in wireless mode) and listening for a response instead of waiting for a broadcast beacon packet.

5.2.3 Architectural improvements

When it comes to optimizing the underlying architecture of the tool, there are some points where it could be improved upon in the future. One avenue for improvement is memory management. The tool clears up old packet pairs but never deletes old connections, flows, or packet headers. Since the tool is designed to run for long periods of time, recording a

number of flows and packets, it is important that it does not gradually fill up the underlying hardware's limited memory with old, irrelevant information. A possible solution could be to store previous packets in a database such as MongoDB, which can easily store Python objects such as Scapy packets and keep only a few whole packet headers in memory. This would allow the user to go back to any time and have the tool analyze and compare packets from around that time without keeping it all in memory.

The GUI, as it stands, is only able to run on the same device as the one running the tool (a Raspberry Pi), meaning that users currently have three ways of viewing it, none of which are ideal. They can either connect a display to the device, save the packet capture for later offline viewing on another device, or use VNC software to view the desktop of the device remotely, which is usually slow and could make for a better user experience. Since the idea is for the device to be set up and run without much physical user interaction or maintenance, allowing the GUI to run remotely from the device would be a better alternative. This could be done by porting the GUI to run in a web application that can be accessed from any other device, or sending the data from the monitoring device to another device running the GUI locally. The last solution may not be too much work to implement as the plotting part of the GUI already runs in a separate process.

Lastly, it might prove beneficial to incorporate some functionality similar to ePPing, which utilizes eBPF to run much of the processing in kernel space and, as a result, runs much more efficiently and is able to process a lot more data.

5.2.4 Future tests

There are some experiments that would be interesting to run, which might yield some relevant results. One example is to experiment with signal strength. This test could be done by placing wireless devices at different positions around the AP and experimenting with obstructions, such as walls, or interference from other APs and devices on the same wireless channel. Looking at data rates in relation to airtime fairness, especially if the AP supports fair queuing, could say a lot about the state of the WLANs.

Bibliography

[1] Vijay Kumar Adhikari et al. 'Unreeling netflix: Understanding and improving multi-CDN movie delivery'. In: *2012 Proceedings IEEE INFOCOM*. 2012, pp. 1620–1628. DOI: 10.1109/INFCOM.2012.6195531.

[2] *Aircrack-ng FAQ*. URL: https://www.aircrack-ng.org/doku.php?id=faq#what_is_the_best_wireless_card_to_buy (visited on 19/12/2022).

[3] Tom Barbette et al. 'Cheetah: A High-Speed Programmable Load-Balancer Framework With Guaranteed Per-Connection-Consistency'. In: *IEEE/ACM Transactions on Networking* 30.1 (2022), pp. 354–367. DOI: 10.1109/TNET.2021.3113370.

[4] Petter Juterud Barhaugen. *NETHINT source code*. https://github.com/petternett/NETHINT. 2023.

[5] David Borman et al. *TCP Extensions for High Performance*. RFC 7323. Sept. 2014. DOI: 10.17487/RFC7323. URL: https://www.rfc-editor.org/info/rfc7323.

[6] *Bufferbloat.net — Projects*. https://www.bufferbloat.net/projects/. (Visited on 11/02/2023).

[7] Deloitte. *Digital Consumer Trends 2020 - The Nordic Cut*. Tech. rep. Deloitte, 2020. URL: https://www2.deloitte.com/content/dam/Deloitte/se/Documents/technology-media-telecommunications/deloitte-digital-consumer-trends-2020-nordic.pdf.

[8] Nandita Dukkipati et al. *Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses*. Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01. Work in Progress. Internet Engineering Task Force, Feb. 2013. 20 pp. URL: https://datatracker.ietf.org/doc/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[9] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293.

[10] Gorry Fairhurst and Colin Perkins. *Considerations around Transport Header Confidentiality, Network Operations, and the Evolution of Internet Transport Protocols*. RFC 9065. July 2021. DOI: 10.17487/RFC9065. URL: https://www.rfc-editor.org/info/rfc9065.

[11] Simone Ferlin-Reiter et al. 'Revisiting Congestion Control for Multipath TCP with Shared Bottleneck Detection'. In: Jan. 2015. DOI: 10.1109/INFOCOM.2016.7524599.

[12] Sally Floyd, Dr. K. K. Ramakrishnan and David L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001. DOI: 10.17487/RFC3168. URL: https://www.rfc-editor.org/info/rfc3168.

[13] Matias Fontanini. *dot11decrypt*. https : / / github . com / mfontanini / dot11decrypt. 2017.

[14] Matthew Gast. *802.11ac: A Survival Guide*. 1st. O'Reilly Media, Inc., 2013. ISBN: 1449343147.

[15] Sofiane Hassayoun, Janardhan Iyengar and David Ros. 'Dynamic Window Coupling for multipath congestion control'. In: *2011 19th IEEE International Conference on Network Protocols*. 2011, pp. 341–352. DOI: 10.1109/ICNP.2011.6089073.

[16] M. Heusse et al. 'Performance anomaly of 802.11b'. In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*. Vol. 2. 2003, 836–843 vol.2. DOI: 10.1109/INFCOM.2003.1208921.

[17] Toke Høiland-Jørgensen, Per Hurtig and Anna Brunstrom. 'The Good, the Bad and the WiFi: Modern AQMs in a residential setting'. In: *Computer Networks* 89 (2015), pp. 90–106. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2015.07.014. URL: https://www.sciencedirect.com/science/article/pii/S1389128615002479.

[18] Toke Høiland-Jørgensen et al. 'Ending the Anomaly: Achieving Low Latency and Airtime Fairness in WiFi'. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 139–151. ISBN: 978-1-931971-38-6. URL: https : / / www . usenix . org / conference / atc17 / technical - sessions / presentation/hoilan-jorgesen.

[19] Toke Høiland-Jørgensen et al. *The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm*. RFC 8290. Jan. 2018. DOI: 10.17487/RFC8290. URL: https://www.rfc-editor.org/info/rfc8290.

[20] *How do I turn on Air Time Fairness on my access point?* https : / / kb . netgear.com/000060680/How-do-I-turn-on-Air-Time-Fariness-on-my-access-point. 2020. (Visited on 02/05/2023).

[21] 'IEEE Standard for Ethernet'. In: *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)* (2022), pp. 1–7025. DOI: 10 . 1109 / IEEESTD . 2022 . 9844436.

[22] 'IEEE Standard for Information technology - Local and metropolitan area networks - Part 3: CSMA/CD Access Method and Physical Layer Specifications - Media Access Control (MAC) Parameters, Physical Layer, and Management Parameters for 10 Gb/s Operation'. In: *IEEE Std 802.3ae-2002 (Amendment to IEEE Std 802.3-2002)* (2002), pp. 1–544. DOI: 10.1109/IEEESTD.2002.94131.

[23] '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)* (2021), pp. 1–767. DOI: 10.1109/IEEESTD.2021.9442429.

[24] 'IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications'. In: *IEEE Std 802.11-1997* (1997), pp. 1–445. DOI: 10.1109/IEEESTD.1997.85951.

[25] *Internet Architecture Board — Measuring Network Quality for End-Users, 2021*. https://www.iab.org/activities/workshops/network-quality/. 2021. (Visited on 30/04/2023).

[26] Marco Iorio, Fulvio Risso and Claudio Casetti. 'When Latency Matters: Measurements and Lessons Learned'. In: *SIGCOMM Comput. Commun. Rev.* 51.4 (Dec. 2021), pp. 2–13. ISSN: 0146-4833. DOI: 10.1145/3503954.3503956. URL: https://doi.org/10.1145/3503954.3503956.

[27] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000.

[28] Hao Jiang and Constantinos Dovrolis. 'Passive Estimation of TCP Round-Trip Times'. In: *SIGCOMM Comput. Commun. Rev.* 32.3 (July 2002), pp. 75–88. ISSN: 0146-4833. DOI: 10.1145/571697.571725. URL: https://doi.org/10.1145/571697.571725.

[29] *Kernel.org — Segmentation offloads*. https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html. (Visited on 15/04/2023).

[30] Steve McCanne. *libpcap: An Architecture and Optimization Methodology for Packet Capture*. 2011. URL: https://sharkfestus.wireshark.org/sharkfest.11/presentations/McCanne-Sharkfest'11_Keynote_Address.pdf.

[31] Steven McCanne and Van Jacobson. 'The BSD Packet Filter: A New Architecture for User-Level Packet Capture'. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, 1993, p. 2. URL: https://dl.acm.org/doi/10.5555/1267303.1267305.

[32] *nginx home page*. https://nginx.org/en/. 2023. (Visited on 09/05/2023).

[33] Kathleen Nichols. *PPing*. 2018. URL: https://github.com/pollere/pping.

[34] Sumiyo Okada et al. 'Detecting Wireless LAN Bottlenecks Using TCP Connection Measurement at Traffic Aggregation Point'. In: *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 2019, pp. 1–6. DOI: 10.23919/APNOMS.2019.8892945.

[35] Christoph Paasch et al. *Responsiveness under Working Conditions*. Internet-Draft draft-ietf-ippm-responsiveness-02. Work in Progress. Internet Engineering Task Force, Mar. 2023. 24 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-ippm-responsiveness/02/.

[36] Giorgos Papastergiou et al. 'De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives'. In: *IEEE Communications Surveys Tutorials* 19.1 (2017), pp. 619–639. DOI: 10.1109/COMST.2016.2626780.

[37] Raviraj Ganesh Rathor and Radhika D. Joshi. 'Performance Analysis of IEEE802.11ax (Wi-Fi 6) Technology using Multi-user MIMO and Up-Link OFDMA for Dense Environment'. In: *2021 IEEE 2nd International Conference on Applied Electromagnetics, Signal Processing, & Communication (AESPC)*. 2021, pp. 1–7. DOI: 10.1109/AESPC52704.2021.9708544.

[38] Dan Rubenstein, Jim Kurose and Don Towsley. 'Detecting Shared Congestion of Flows via End-to-End Measurement'. In: *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '00. Santa Clara, California, USA: Association for Computing Machinery, 2000, pp. 145–155. ISBN: 1581131941. DOI: 10.1145/339331.339410. URL: https://dl.acm.org/doi/10.1145/339331.339410.

[39] Jan Ruth and Oliver Hohlfeld. 'Demystifying TCP Initial Window Configurations of Content Distribution Networks'. In: *2018 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, June 2018. DOI: 10.23919/tma.2018.8506549. URL: https://doi.org/10.23919/tma.2018.8506549.

[40] Marta Rybczyńska. *A QUIC look at HTTP/3*. Mar. 2020. URL: https://lwn.net/Articles/814522/ (visited on 29/03/2023).

[41] Matt Sargent et al. *Computing TCP's Retransmission Timer*. RFC 6298. June 2011. DOI: 10.17487/RFC6298. URL: https://www.rfc-editor.org/info/rfc6298.

[42] *Scapy home page*. https://scapy.net/. 2023. (Visited on 12/05/2023).

[43] Stanislav Shalunov et al. *Low Extra Delay Background Transport (LEDBAT)*. RFC 6817. Dec. 2012. DOI: 10.17487/RFC6817. URL: https://www.rfc-editor.org/info/rfc6817.

[44] ESnet Software. *iPerf Source Code*. https://github.com/esnet/iperf/blob/master/src/iperf_udp.c. 2022.

[45] W. Richard Stevens. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001. Jan. 1997. DOI: 10.17487/RFC2001. URL: https://www.rfc-editor.org/info/rfc2001.

[46] W. Richard Stevens and Gary R. Wright. *TCP/IP Illustrated (Vol. 2): The Implementation*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 020163354X.

[47] stryngs. *pyDot11*. https://github.com/stryngs/pyDot11. 2022.

[48] Simon Sundberg et al. 'Efficient Continuous Latency Monitoring with eBPF'. In: *Passive and Active Measurement*. Ed. by Anna Brunstrom, Marcel Flores and Marco Fiore. Cham: Springer Nature Switzerland, 2023, pp. 191–208. ISBN: 978-3-031-28486-1.

[49] Martin Thomson and Sean Turner. *Using TLS to Secure QUIC*. RFC 9001. May 2021. DOI: 10.17487/RFC9001. URL: https://www.rfc-editor.org/info/rfc9001.

[50] Brian Trammell et al. 'Enabling Internet-Wide Deployment of Explicit Congestion Notification'. In: *Passive and Active Measurement*. Ed. by Jelena Mirkovic and Yong Liu. Cham: Springer International Publishing, 2015, pp. 193–205. ISBN: 978-3-319-15509-8.

[51] *Transmission Control Protocol*. RFC 793. Sept. 1981. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/info/rfc793.

[52] Michael Welzl. *Network Congestion Control: Managing Internet Traffic*. Wiley, Sept. 2005.

[53] Welzl, Islam Teymoori and Gjessing Hutchinson. 'Future Internet Congestion Control: The Diminishing Feedback Problem'. In: (2022). DOI: 10.48550/arXiv.2206.06642. arXiv: 2206.06642 [cs.NI].

[54] *WPA3 Specification*. Tech. rep. Version 3.1. Wi-Fi Alliance, Nov. 2022. URL: https://www.wi-fi.org/downloads-public/WPA3%2BSpecification%2Bv3.1.pdf/35332.

[55] Wei Yin et al. 'MAC-layer rate control for 802.11 networks: a survey'. In: *Wireless Networks* 26.5 (July 2020), pp. 3793–3830. ISSN: 1572-8196. DOI: 10.1007/s11276-020-02295-2. URL: https://doi.org/10.1007/s11276-020-02295-2.

[56] S. Zander and G. Armitage. *Minimally-Intrusive Frequent Round Trip Time Measurements Using Synthetic Packet-Pairs - Extended Report*. Melbourne, VIC, 2013. URL: http://caia.swin.edu.au/reports/130730A/CAIA-TR-130730A.pdf.

[57] Sebastian Zander and Grenville Armitage. 'Minimally-intrusive frequent round trip time measurements using Synthetic Packet-Pairs'. In: *38th Annual IEEE Conference on Local Computer Networks*. 2013, pp. 264–267. DOI: 10.1109/LCN.2013.6761245.