Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study programme/specialisation: Bachelor in Computer Science	Spring semester 2023 Open
Authors: Jakob Bernhardt Danielsen, Endre Lund and Mats Husberg	
Program coordinator: Erlend Tøssebro Supervisor: Naeem Khademi Co-Supervisor: Aitor Martin Rodriguez	
Title of bachelor's thesis: Online Gaming Performance Evaluation over Starlink Satellite Broadband	
Credits: 20	
Keywords: Satellite, Online gaming, Starlink	Number of pages: 69 + supplemental material/other: 58 Stavanger 15. mai 2023

University
of Stavanger

JAKOB BERNHARDT DANIELSEN, ENDRE LUND AND MATS HUSBERG
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Online Gaming Performance Evaluation over Starlink Satellite Broadband

Bachelor's Thesis - Computer Science - May 2023

I, **Jakob Bernhardt Danielsen, Endre Lund and Mats Husberg**, declare that this thesis titled, “Online Gaming Performance Evaluation over Starlink Satellite Broadband” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a bachelor’s degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Abstract

This thesis studies the performance and the Quality of Experience (QoE) of online gaming over the Starlink satellite network and compares it to terrestrial access technologies. Satellite broadband provides an opportunity to play games online from areas where traditional access networks are insufficient or not available. Modern games require certain standards of network performance, and games of different game genres have different requirements for certain metrics. Said metrics and how Starlink performs are investigated in this thesis. Our work provides results from experimental scenarios in two games from different game genres. The objective of this work is to evaluate the performance of online gaming through Starlink, and if the QoE meets the expectations of players.

Acknowledgements

We would like to thank our supervisor Associate Professor Naeem Khademi and co-supervisor Aitor Martin Rodriguez for their enthusiasm and help with the work behind this thesis. We would also thank Ståle Freyer for setting up the Starlink and lab environment, and Theodor Ivesdal for helping with networking and security. Further, we thank our families for their support while writing this thesis.

Contents

Abbreviations

ACK	Acknowledgement
SYN	Synchronize
BBR	Bottleneck-Bandwidth and Round-Trip Time
CBR	Constant Bit Rate
CCA	Congestion Control Algorithm
CLI	Command Line Interface
CS:GO	Counter Strike: Global Offensive
CS model	Client-Server model
cwnd	Congestion Window
IP	Internet Protocol
ISP	Internet Service Provider
FPS	First Person Shooter
GEO	Geostationary Orbit
LEO	Low Earth Orbit
MEO	Medium Earth Orbit
MMOG	Massively Multiplayer Online Game
P2P	Peer-to-Peer
PEP	Performance Enhancing Proxy
QoE	Quality of Experience
QoS	Quality of Service
RTT	Round Trip Time
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VBR	Variable Bit Rate

Chapter 1

Introduction

1.1 Motivation

Broadband has become a requirement in most households and businesses around the world. Generally, broadband is associated with high speeds and continuous connectivity. Fiber-based networks are still recognized as the best option for broadband today [9].

A challenge of fiber-based broadband and other terrestrial connections is coverage. How can high-speed broadband be provided to even the most rural of areas? In an analysis carried out in 2018, [5], Briglauer and Gugler suggested that a 100% penetration of fiber-based broadband is unrealistic. In the paper, the cost-benefit analysis estimated a 50% penetration to be most beneficial for growth in GDP due to the increasingly high costs of rolling out fiber to rural areas.

The growth in demand for high-speed internet and the coverage restraints on the terrestrial network have led to SpaceX developing Starlink, a satellite-based broadband service. The service aims to provide high-speed internet access to its users all around the world through a constellation of Low-Earth-Orbiting (LEO) satellites. Starlink promises to deliver comparable internet speeds and latency comparable to that of terrestrial connections, even to households and businesses in rural areas where traditional broadband services are not available or reliable. This technology can potentially revolutionize how we access and use the internet, especially for gaming enthusiasts. Online gaming has established itself as a mainstream form of entertainment, with millions of people playing online games every day.

The valued gaming market was at USD 203.2 billion in 2021 [38], and projects a market growth at a Compound Annual Growth Rate (CAGR) of 13.4% from 2023 to 2030. With the base of online game players projected to grow, so will the demand for high-speed connections with low latency and minimal packet loss. Starlink promises to provide gamers with a fast and reliable internet connection, even in ”remote and rural locations across the globe” [41].

1.2 Problem Definition

Satellite performance for gaming often offers high latency, packet loss, and low bandwidth, which can cause poor performance and negatively impact the QoE for gaming users. Starlink’s new solution for satellite networks should bring the latency, packet loss, and bandwidth to an adequate level for online gaming.

1.2.1 Research Questions

For this thesis, we will dive deeper with these research questions:

1. Does Starlink satisfy latency and bandwidth requirements for online gaming?
2. Does Starlink produce significant packet loss with regard to online gaming?
3. What are the differences in performance between terrestrial internet access and Starlink?

1.3 Objectives

The goal is to set up a Starlink antenna/dish on the roof of the University of Stavanger and create automated scripts to provide some results to analyze. Firstly performing a baseline measurement before investigating deeper into the gaming part.

After defining the research questions, more objectives of the work are:

1. Study how satellite performance is related to the metrics of gaming: Latency, jitter, packet loss, and bandwidth
2. Set up a testbed to run automated experiments simultaneously on both server and client
3. Create an automated testbed for baseline experiments of the Starlink connection
4. Obtain results of performance over the Starlink connection
5. Discuss the results obtained and reflect on the impact it has on the satellite gaming performance

1.4 Outline

This thesis has the following structure:

- Chapter 2 describes the differences in satellite solutions, and online gaming with insight into the games used in this thesis and key metrics for performance evaluation
- Chapter 3 shows the methodology of the work done, showing and describing the different tools used and the scenarios
- Chapter 4 is where the results are presented and how the results are achieved.
- Chapter 5 discusses the results given in Chapter 4 and how the results compare to the theory behind it.
- Chapter 6 is the conclusion, where we summarize and conclude our findings in this thesis and answer the research questions given in the introduction

Chapter 2

Background

The background aims to provide context for the work done in this thesis. It describes the foundation set by related works in the field, and it discusses its potential importance for the solution proposed for the problem statement.

This chapter starts with a section discussing satellite as a broadband service. It introduces three of the more common orbits: the Geostationary Orbit (GEO), the Medium Low Orbit (MEO), and the LEO. Then it describes their characteristics and the implications those have for coverage and network performance.

The section then introduces Starlink satellite broadband. Here, a brief overview of the constellation is given. Next, benchmarks of Starlinks network performance are provided, giving insight into whether the service has the potential to deliver what is necessary for online gaming. The section ends by introducing a possible problem for Starlink when online gaming is concerned; satellite handovers could induce spikes in the form of both packet loss and jitters. This could in turn affect player experience.

The next section in this chapter introduces online gaming. The start of this section explains what online gaming is and how it started, before introducing different online gaming architectures and genres. Next, the two games through which this thesis aims to test Starlink broadband are brought up. Related works on the games' demand for network performance are then presented.

Ending the chapter is a summary of related works. This section aims to organize the related works and present research gaps that our thesis potentially can fill.

2.1 Satellite Broadband Service

Satellites are generally categorized by which orbit they are traveling in. The three most common orbits are GEO, MEO, and LEO. Said categorizations are determined by altitude as seen in table 2.1.

Type of orbit	**Altitude**
GEO	35 786 km
MEO	Between GEO and LEO
LEO	<1000 km

Table 2.1: Types of orbits and their altitudes per The European Space Agency [3]

With the rise in demand for high-speed internet, broadband provided by modern communications satellites has emerged as a potential solution to the lack of coverage provided by terrestrial connections.

2.1.1 Satellite types

2.1.1.1 GEO satellites

Satellites in the geostationary orbit move at a speed of around 3 kilometers per second at an altitude of 35 786 kilometers [3]. An advantage to orbiting at such a high altitude is coverage, and it is estimated that three GEO satellites can provide close to global coverage. Some GEO satellites provide up to 100 Mbit/s on the downlink for the end user [11]. GEO satellite broadband is a good option for those in rural areas, though not for applications requiring a low propagation delay. In the best-case scenario, a propagation delay of at least 240ms is achieved [10], though together with other delays, a common Round Trip Time (RTT) for GEO satellites is roughly 600ms.

To mitigate the high delays, commercial satellite networks often use Performance Enhancing Proxies (PEPs) [11]. A characteristic of PEPs is the varying degree to which it is transparent to a network's end systems or to applications. It could turn out that Starlink Satellite Broadband does use PEPs, but not network-layer transparent PEPs [17]. This would mean that neither the Transmission Control Protocol (TCP)/Internet Protocol (IP) stack nor the applications would be aware of the PEP implementation.

2.1.1.2 LEO satellites

Satellites in the LEO circle the earth in different planes, normally at altitudes less than 1000 km but also as low as 160 km [3]. At such altitudes, LEO satellites cover small areas of the earth at once. They are typically launched in groups (i.e. a constellation), and thousands of them can provide global coverage.

For this to work, antennas need to be steerable and there need to be frequent satellite handovers [10]. On the other hand, LEO satellites provide propagation delays of only just a few milliseconds. Deutschmann et al. [11] propose that LEO mega-constellations have the potential to provide low latencies and high data rates for broadband internet.

2.1.1.3 MEO satellites

The MEO is a wide range of orbits anywhere between GEO and LEO [3]. Deutschmann et al. [11] suggest that an MEO satellite constellation at an altitude of about 8000 km provides a trade-off between the number of deployed satellites and acceptable latencies. MEO satellites are commonly used for navigation.

2.1.2 Starlink

Starlink is today the largest satellite constellation in the LEO. According to [42], 4165 satellites have been launched into space as of April 20th of this year. Of them, 3423 are active, 475 are inactive and 267 are burned. The Starlink satellites use an altitude of about 550 km [10]. Starlink [43] claims that their constellation can "deliver broadband internet capable of supporting streaming, online gaming, video calls and more".

2.1.2.1 Starlink Performance

Michel et al. [26] performed a benchmark on the Starlink service. It measured throughput for QUIC and TCP, packet loss, and latency. In addition, the study measured the QoE for Web browsing with Starlink. The latency was measured in two ways; with and without load on the link. They found the minimum latency of Starlink to be around 20ms for close destinations, rising to as high as a few hundred milliseconds under traffic load at sub-optimal destinations. For packet loss, they found that loss occurs more frequently under traffic load, but that it only affects a few consecutive packets. Without traffic load, the loss occurs less frequently but affects more consecutive packets and lasts longer.

The study then found that Starlink's download throughput ranges between 100 to 250 Mbit/s, with a median value of 178 Mbit/s. The upload throughput median value they found was 17 Mbit/s. Lastly, Michel et al. found that Starlink outperforms traditional Satellite Communication for Web browsing; comparable to that of regular wired access.

The same study [26] also analyzed the Starlink network for a potential presence of PEPs, middleboxes, and traffic discrimination (TD). They did not find any presence of any PEPs. The same was the outcome regarding traffic discrimination, as no TD policy was found.

”Starlinkstatus.space” [46] collects community Starlink performance data from different regions in the world. The site provides statistics on ICMP Ping and available bandwidth for the downlink (DL) and for the uplink (UL), as seen in table 2.2.

Region	**Latency**	**DL throughput**	**UL throughput**
Worldwide	47ms	157 Mbit/s	14 Mbit/s
USA	49ms	127 Mbit/s	11 Mbit/s
EU	45ms	206 Mbit/s	16 Mbit/s

Table 2.2: Average Starlink performance values rounded to the nearest whole number.

2.1.2.2 Satellite Handovers

A potential obstacle for Starlink satellite broadband for online gaming is a potential Quality of Service (QoS) degradation due to a satellite handover. When a current satellite goes out of sight, the satellite dish needs to switch the connection to another satellite. Kassem et al. [22] found significant rates of sudden User Datagram Protocol (UDP) packet loss in correlation with the satellites going out of the line of sight, strongly suggesting that the satellite switching causes severe packet loss. The study goes on to suggest that congestion control algorithms that are not loss-based could be an option to enhance Starlink's performance.

2.2 Online Gaming

Online gaming is video games being played by players in different locations connected together through the Internet. The players interact with a game world (state) that needs to be maintained by either a server or the personal computers of the players themselves.

The first online game was created when outside users connected to a text-based dungeon adventure game called MUD [34]. The game was developed by two undergraduate students at the University of Essex in the year 1980. This laid the foundation for expansion from other programmers, and it further led to the first wave of Massively Multiplayer Online Games (MMOG) introduced in the late 90s [34]. Together with the expansion of broadband internet connectivity in the early 2000s, online gaming became a popular form of entertainment for millions of people.

2.2.1 Online Gaming Architecture

When online gaming transitioned from being deployed in local area networks to wide-area networks, two main underlying online game architectures were and remain the favored alternatives. They are depicted in figure 2.1.

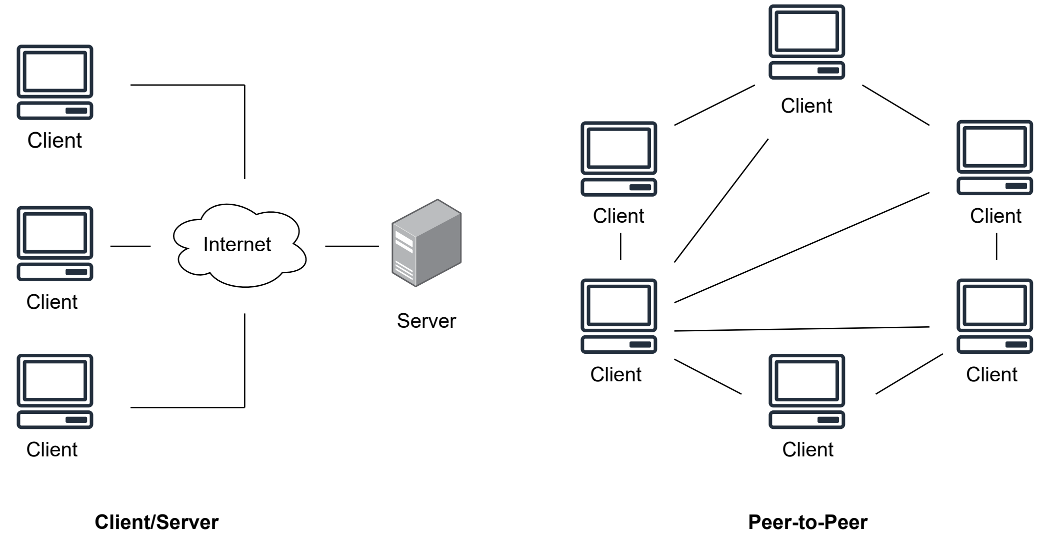


Figure 2.1: Client-Server model (CS model) vs Peer-to-Peer (P2P) comparison

2.2.1.1 Peer-to-Peer

P2P is a serverless online gaming architecture where game clients connect to each other's PCs in order to play in a multiplayer setting. Each client in-game has to maintain the state of the game, which leads to a greater demand for processing and memory on client PCs compared to server-based architectures.

Though P2P is not a popular choice among developers today, it is still a favorable choice for multiplayer indie games due to low costs [39]. Examples of popular P2P games are GTA Online [15] and Super Smash Bros. Ultimate [30].

2.2.1.2 The Client-Server Model

CS model is a centralized online gaming architecture where a dedicated server maintains the virtual environment and provides real-time world updates to its connected client PCs. It is by far the most popular choice for game developers as it is more secure and more scalable than any other architecture. The CS model is also the online gaming architecture that is used in both of the games that this thesis is evaluating gaming performance on.

2.2.2 Online Gaming Genres

There are many different genres online games fall into. Of them, only two are considered in this thesis.

2.2.2.1 Massively Multiplayer Online Games

MMOGs are games that are played by a huge amount of players at the same time. In MMOGs the virtual environment is persistent, meaning the game world cannot be stopped. If players disconnect from the game server, the game world continues without them. In networking terms, this means the server runs continuously and that clients can connect to the persistent game world at any time. Examples of widely popular MMOGs are Word of Warcraft [12] and Runescape [25].

2.2.2.2 First Person Shooter

First Person Shooters (FPS) are combat-oriented shooter games played from a first-person perspective. Except for Massively Multiplayer Online FPS, FPS games are not persistent but rather played in iterations of time-limited matches. The most common game type of these matches is deathmatches, a game mode where players gain points by killing other player characters. Call of Duty [1] and Valorant [14] are two examples of popular FPS games today.

2.2.3 Minecraft

The first game that was picked to evaluate the Starlink Broadband is the MMOG Sandbox game Minecraft [27]. Minecraft was released in 2009 and is a very popular adventure and construction-based game. The game has a CS model architecture.

2.2.3.1 Transport Protocol

From any active Minecraft server's details description, it says that all Minecraft game traffic uses TCP. Any client-to-server connections must first establish the connection, and they both keep track of all packets and their sequence of them. If a packet is lost in transmission, the packet will be re-sent as a TCP Retransmission. This essentially means that packet loss is not as relevant for TCP-based games (as opposed to UDP-based games), but rather that packet loss induces more latency.

2.2.3.2 Metrics

Minecraft is in general much less sensitive to Quality of Service degradation compared to FPS games. Hohlfeld et al. [19] carried out an experiment where casual gamers played Minecraft with induced delays of 0ms, 170ms, and 1000ms respectively. The study did not find the effects of said latencies to be statistically significant, but rather barely visible. Another study [4] found that delay will impact gameplay from 250ms onwards.

2.2.4 Counter Strike: Global Offensive

The second game that was picked to evaluate the Starlink Broadband is the FPS game Counter Strike: Global Offensive (CS:GO) [8], a very popular shooter released in 2012. The game has a Client/Server architecture.

2.2.4.1 Transport Protocol

According to a study done in 2020 regarding CS:GO [18], the transport protocol for this game is UDP. A characteristic of CS:GO is high interactivity with low latency. This essentially means that the game is generating a large number of packets per second, also referred to as the tick rate. CS:GO's official servers run on 64 ticks, however, third-party services can host game servers with a tick rate of up to 128.

2.2.4.2 Metrics

CS:GO is an FPS game, and as mentioned in 3.1, it has the highest limits for latency for any game genre. Xu et al. [51] found latency in CS:GO to be significantly impacting player performance. Their results suggest an additional 100ms of delay reduces a player's shooting accuracy with an AK-47 assault rifle by about 15%. They also found a reduction of QoE of about 11% with a delay of 100ms. Quax et al. [33] concluded that an FPS player's QoE relies on the size of latency the network introduces, and found delay jitters below 100ms to be hampering the experience. The study did however show there are indications of performance degradation from a delay of 60ms and onwards. Beigbeder et al. [6] found that shooting mechanics was ”greatly affected” by latencies in a range of 75ms to 100ms and that shooting accuracy and the number of kills decrease up to 50% for such latency ranges. The study found that experiment subjects noticed latency as low as 100ms and that latency of 200ms is annoying.

A direct consequence of using UDP as the transport protocol is a higher degree of packet loss. The priority of CS:GO is high interactivity and low latency. Beigbeder et al. [6] did however find that packet loss did not have any measurable effect on player performance. The study then explained that users could barely even notice a packet loss of 5%.

2.3 Summary of Related Works

Below are related works summarized in two tables: Satellite Broadband 2.3 and Performance Demands of Online Gaming 2.4.

2.3.1 Satellite Broadband

Src	**Traffic type(s)**	**Metric(s)**	**Findings**	**Research gap**
[22]	TCP	Throughput, Loss	Throughput: high geographical difference. Highest median throughput: 147Mbit/s. Lowest: 34.3Mbit/s. Loss: Rare spikes of up to 50% loss. 12% of iPerf tests had a loss over 5%. Loss strongly correlates with satellite handovers.	Lacks RTT measurements. Does not evaluate UDP traffic. Does not test gaming over Starlink.
[10]	TCP	Goodput, Latency, Loss	Median goodput around 185Mbit/s. Latency ”usually did not exceed 50ms”. Packet loss of about 1.8%	Does not evaluate UDP traffic. Does not test gaming over Starlink.
[26]	QUIC, TCP	Throughput, Latency, Loss, QoE for Web browsing	Minimum delay 20ms. Maximum a few hundred ms under traffic load at suboptimal locations. Median downlink throughput 178Mbit/s. Median uplink throughput 17Mbit/s. Starlink outperforms traditional SatCom for Web browsing. QUIC packet loss: 1.56% on the downlink and 1.96% on the uplink. Found no presence of PEPs.	Does test Web browsing over Starlink, but does not test gaming.
[11]	GEO, LEO	Bandwidth	Some GEO satellites provide up to 100Mbit/s. They usually use PEPs to mitigate high delays. The high latencies with GEO satellites are problematic for some applications. LEO mega-constellations have the potential for low delays and high data rates.	Does not conclude whether the LEO broadband solution is applicable for gaming.

Table 2.3: Table of related works on satellite broadband

Related works on satellite broadband have found that although GEO satellites provide enough bandwidth for online gaming, latencies are too high. Starlink latencies are found to be promising for the most part, except in certain edge cases.

Related works have found Starlink's performance regarding throughput to be sufficient for online gaming purposes. Packet loss is found to be strongly correlated with satellite handovers, which might pose a problem for online gaming.

Although works have been done on QoE for web browsing over Starlink, no previous works cover gaming over LEO satellite broadband. This thesis aims to cover this research gap.

2.3.2 Performance demands of Online Gaming

Src	**Game (Genre)**	**Metric(s)**	**Findings**	**Research gap**
[4]	Minecraft (MMOG)	Delay	Minecraft gameplay experience degrades with delays over 250ms	Does not test for experience degradation due to packet loss. It is not performed over Starlink.
[19]	Minecraft (MMOG)	Delay	Delays up to 1000ms in Minecraft does not degrade the player experience and is barely visible.	It is not performed over Starlink.
[35]	(MMOG)	Delay, Loss, Jitters	In MMOGs, increased latency negatively impacts subjective quality, more so than jitters. MMOGs were less reliant on delay than FPS games. Packet loss leads to a strong reduction in perceptual quality.	It is not performed over Starlink.
[33]	(FPS)	Latency, Jitters	Indications of performance degradation with delays over 60ms for FPS games. Jitters below 100ms are found to be hampering the experience.	It is not performed over Starlink.
[51]	CS:GO (FPS)	Delay, QoE	Additional 100ms of delay reduces a player's shooting accuracy with an AK-47 assault rifle by about 15%. QoE down 11% with 100ms delay.	It does not test player experience over Starlink.
[6]	(FPS)	Delay, Loss	Shooting mechanics was "greatly affected" by latencies in a range of 75ms to 100ms, and that shooting accuracy and the number of kills decrease up to 50% for such latency ranges. 100ms delay noticable, 200ms delay annoying. packet loss did not have any measurable effect on player performance. 5% was barely noticeable.	It does not test network performance playing FPS games over Starlink.
[29]	(FPS)	Delay	Found that the FPS genre has the tightest latency limits.	It does not test player experience over Starlink.

Table 2.4: Table of related works on network demands of online gaming

Related works on online gaming have found that the FPS game genre has the tightest latency limits, with MMOGs being less reliant on latency.

There are related works on QoS sensitivity for both Minecraft and CS:GO. There is no related work that evaluates to which degree the QoS for both games is satisfactory over the Starlink broadband. This thesis aims to cover this research gap.

Chapter 3

Methodology

This chapter describes the different testbed implementations, hardware equipment, various software and tools, post-processing, and other setup configurations.

3.1 Game Selection

Different online games have different network parameter demands. Thus, the choice of games to test in our work is important. First, we decided to pick an FPS, as this genre is considered to have the tightest latency limits [29].
Secondly, we picked an MMOG Sandbox game. The Sandbox game genre represents an open game world where the player can interact freely with the world. For MMOGs, a study [35] found that increased latency negatively impacts subjective quality, more so than jitters. Still, MMOGs are less reliant on latency than FPSs.

Most real-time interactive games use UDP as their transport protocol. We decided to pick two widely popular games to test with: one that uses UDP and one that uses TCP.

3.2 Testbed overview

The testbed used for this thesis is rewired in two different ways, each depending on which task to execute. The topology for the different testbed setups/networks that were used is shown in figure 3.1 and 3.2.

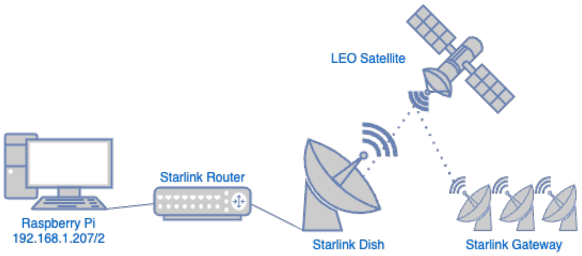


Figure 3.1: Topology describing the network used for latency measurements

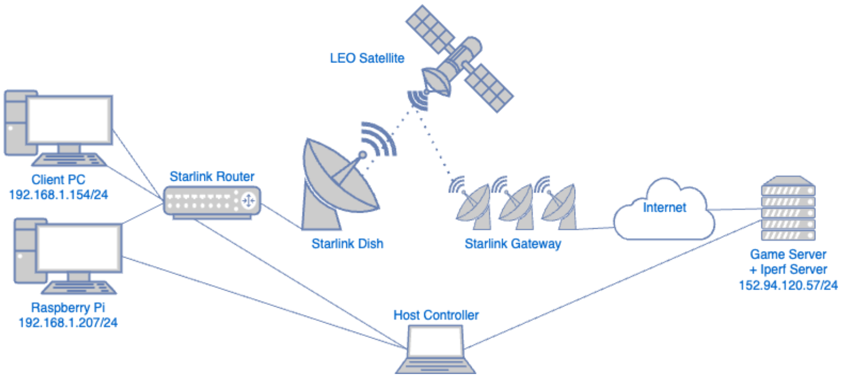


Figure 3.2: Topology describing the network used regarding iPerf3 measurements and for capturing game traffic

- The Host Controller takes care of remotely accessing the various nodes and executing commands or scripts on them
- The Host Controller uses Secure Shell (SSH) protocol to access the various machines
- The Raspberry Pi is used for more lightweight baseline measurements

3.3 Hardware Equipment

3.3.1 Server

To host games and other services, we used an HP Compaq 8200 Elite as a server with specifications listed in table 3.1.

CPU	4-core Intel i5-2400 3.40GHz
RAM	16GB DDR3
Storage	500GB
OS	Debian 11
Kernel-Version	3.38.5

Table 3.1: Server PC specifications

3.3.2 Client

The client PC is mainly used for playing online games, however, it is also used for some light-weight network measurements. Specifications listed in table 3.2.

CPU	Intel core i5-6600K 3.50 GHz
GPU	Geforce GTX 970 4GB
RAM	16GB DDR4 2133 MHz
Storage SSD(OS)	240GB
Storage HDD(Games)	1TB
OS	Windows 10
Kernel-Version	22H2

Table 3.2: Client PC specifications

3.3.3 Raspberry Pi

For some network measurements, we used a Raspberry Pi 4 Model B. This device has the specification listed in table 3.3.

CPU	4-core Arm Cortex A-72 1.50GHz
Ram	4GB DDR4
Storage	32GB
OS	Debian 11
Kernel-Version	3.38.5

Table 3.3: Raspberry Pi 4 model B specifications

3.3.4 Starlink Setup

The Starlink package ordered for this thesis is the baseline residential package [41]. The equipment inside is listed below:

1. One Dish/Antenna
2. One Router
3. One Starlink Cable
4. One AC Cable

The dish consists of an electronic phased array with a 100°field of view. It is placed on top of Kjølv Egelands hus at UiS approximately 63 meters above sea level, see figure 3.3. It weighs in under 3 kg and can handle winds up to 20m/s. The antenna is facing south (180°), with a clear view of ±50°. The antenna communicates with satellites that are visible on the horizon above an elevation angle of 25°.

Figure 3.3: Placement of the Starlink dish.

The antenna is facing south to connect to LEO satellites orbiting over central parts of Europe. Norway does not have a lot of satellite coverage as of the time of this thesis, as shown in figure 3.4.

Figure 3.4: Screenshot of Starlink constellation map, green dot represents our position, white dots satellites, and red dots gateways [40]

3.4 Software

This section seeks to explain the various software and tools used for the thesis. Table 3.4 shows the software and tools used. To do this, the different sections are divided as shown in 3.4 to make experimenting easier.

Section	Title	Includes
3.4.1	Traffic Generators	iPerf, Ping, TCP ping, CS:GO, Minecraft
3.4.2	Loggers	Tshark, iPerf3 logs, Yr weather API, N2YO API
3.4.3	Experiment Automation	Python scripts, Libraries, Diagrams
3.4.4	Post Processing	Analysis, SPP, Visualization
3.4.5	Other Setup Configurations	Security, Difficulties

Table 3.4: Software and tools overview

3.4.1 Traffic Generators

3.4.1.1 iPerf3

iPerf3 is a tool for generating traffic and performing network measurements [21]. It is cross-platform, so it works with Windows, Linux, MacOS, FreeBSD, Android, and more. The default iPerf3 port is 5201.

It is based on IP networks and works with both IPv4 and Ipv6. iPerf uses TCP as the default protocol, however, UDP-specific tests can be instantiated. It can measure total available bandwidth through TCP. Through UDP the client can create streams with specified bandwidth, measure packet loss, and delay jitter.

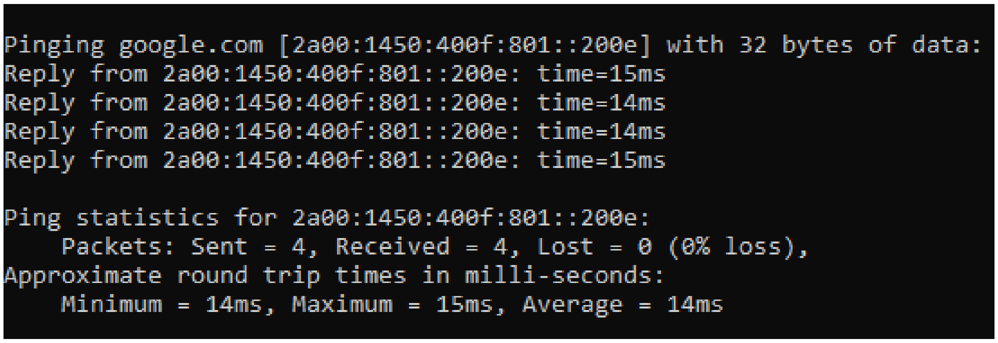
For setup, iPerf uses a client host connection and needs to be installed on both ends, see figure 3.2. On the client side, the IP address needs to be specified along with other inputs that fit the needs of the measurement. See table 3.5 for some common parameters.

Parameter	Description
-p	Specify the port number to listen or connect to (client and host specific)
-s	Run iPerf in server mode (host-specific)
-c	Run iPerf in client mode, connecting to a host (client specific)
-u	Use UDP rather than TCP for tests (client specific)
-b	Set target bandwidth to n bit/s (client specific)
-R	Run in reverse mode, default is uplink client-to-server (client specific)
-t	The time in n seconds to transmit for (client specific)

Table 3.5: Table of common iPerf3 parameters [20]

3.4.1.2 Ping

Ping is a basic open-source internet application that allows users to test if a host is reachable. To reach the destination host users need to know the destination IP address (IPv4 or IPv6) or destination domain name. The way it works is that users send an Internet Control Message Protocol (ICMP) echo request and then wait for a reply. If the echo request reaches the destination host, it answers with an echo reply. By default a ping returns values like RTT, Time To Live (TTL), and averages. If the request does not reach the destination, ping reports the packet as lost. See figure 3.5 for an example output.



```
Pinging google.com [2a00:1450:400f:801::200e] with 32 bytes of data:
Reply from 2a00:1450:400f:801::200e: time=15ms
Reply from 2a00:1450:400f:801::200e: time=14ms
Reply from 2a00:1450:400f:801::200e: time=14ms
Reply from 2a00:1450:400f:801::200e: time=15ms

Ping statistics for 2a00:1450:400f:801::200e:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 14ms, Maximum = 15ms, Average = 14ms
```

Figure 3.5: Example of an ICMP echo request to google.com

3.4.1.3 TCP ping

PsPing's [36] TCP ping uses a slightly different approach than the traditional ICMP echo request and echo reply, where TCP ping uses the TCP protocol to calculate RTT. Utilizing Synchronize (SYN), SYN-Acknowledgement (ACK), and retransmits it can measure the RTT of packets and if a packet has been dropped. So instead of opening a full TCP connection with a three-way handshake, it half-opens the connection and sends an ACK. Then it waits for the SYN-ACK so it can calculate the RTT and close the connection, as shown in figure 3.6. This method can be useful if ICMP packets are blocked by a firewall, and a client wants to measure RTT.

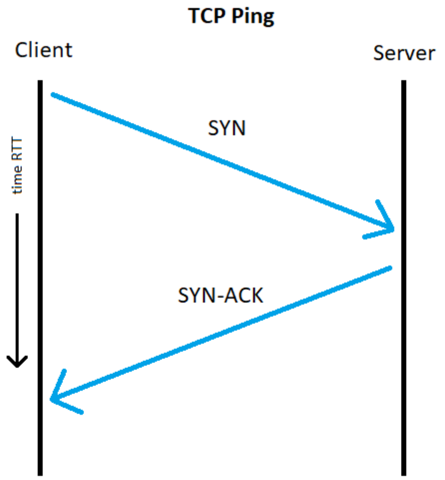


Figure 3.6: TCP ping

3.4.1.4 LinuxGSM

To host games on the Game server 3.1 we used a third-party application called LinuxGSM [24] V23.2.0, which integrates the use of SteamCMD [47] without the need to manage SteamCMD itself. The LinuxGSM software connects to Steam servers via SteamCMD to download game data, in our case CS:GO with appID 730. For Minecraft: Java Edition, LinuxGSM downloaded game data from the official Minecraft site [49].

LGSM is quickly downloaded using on the server using the command:

```
wget -O linuxgsm.sh https://linuxgsm.sh && chmod +x
linuxgsm.sh && bash linuxgsm.sh csgoserver
```

From the command, the CS:GO server is installed with:

```
./csgoserver install
```

Which installs the CS:GO server with appID 730 from Steam. To make the CS:GO server connect to the master server list in-game, we provided it with a valid Game server login token (GLST) from a valid account [44].

The LinuxGSM uses the config files A.1 and A.2 to start up a CS:GO server. The same goes for Minecraft which the server uses the config files A.3 and A.4. These files contain information, parameters, and input that the server uses to start a specific game session.

3.4.1.5 CS:GO

CS:GO is one of the games generating network traffic for this thesis. The transport layer protocol for CS:GO game traffic is UDP. It is installed on the client's PC with default settings. The game version during the time of this thesis is 10905515. The CS:GO server is installed on the game server, see figure 3.1 and sub section 3.4.1.4.

3.4.1.6 Minecraft

Minecraft is the second game generating traffic for this thesis. The transport layer protocol for Minecraft game traffic is TCP. The game is installed on the client's PC with default settings. This includes graphical settings on high and a render distance of 12 chunks. The game version during the time of this thesis is 1.20 Java edition. The Minecraft server is installed on the Game Server, see figure 3.1 and sub section 3.4.1.4.

3.4.2 Loggers

The Raspberry Pi takes care of latency and iPerf3 measurements that run over a longer period. It also logs the results for later post-processing. The client PC and Game Server logs the game traffic and TCP ping measurements.

3.4.2.1 Tshark

Tshark [50] is a network protocol analyzer, which is used to capture data from a live network. The Tshark package is included in the standard Wireshark install. The output is either displayed in a command line interface or written to a file in PCAP format. Tshark works much like the tcpdump tool, however, with Tshark a duration parameter can be used. This makes the code easier and cleaner, which is the reason we chose Tshark and not tcpdump. It can also be used to estimate RTT on captured TCP traffic which needs to be in PCAP format. It is estimated based on the timestamp when a packet is sent and the timestamp of the first ACK. Tshark (version 3.0.2 for Windows and version 2.6.8 for Linux) is installed on both the client PC and the server.

3.4.2.2 iPerf3 Logs

iPerf3-logs output some viable results. These are outputted in intervals that can be set by the host and client. The default interval is 1 second, however, intervals can be set as low as 0.1 seconds. The iPerf logs can be stored as normal text files and accessed later for post-processing.

3.4.2.3 Yr Weather API

To gather real-time weather data for the experiments, we used the API from MET Norway [31] with the parameters for longitude, latitude, and altitude in table 3.6:

Latitude	58.937
Longitude	5.698

Table 3.6: MET Api parameters

We got a JSON response from the API and selected the momentary forecast data to include in our experiments. Here are some of the values that can be extracted from the JSON:

1. Cloud coverage
2. Temperature
3. Wind
4. Precipitation

3.4.2.4 Available Satellites

To measure the distance of satellites relative to a position, an API from N2YO [28] returns what satellites are in view in JSON format as shown in figure 3.7.

The table 3.7 shows the input required for the API.

Parameter	**Type**	**Required**	**Comments**
Latitude	Float	Yes	Ground latitude(decimal degree format)
Longitude	Float	Yes	Ground longitude(decimal degree format)
Altitude	Integer	Yes	Ground altitude above sea level in meter
Search radius	Integer	Yes	Search radius(0-90)
Category ID	Integer	Yes	Id of satellite category

Table 3.7: Required input for the API. Category ID for Starlink is 52.

```
{
    "info": {
        "category": "Starlink",
        "transactionscount": 0,
        "satcount": 18
    },
    "above": [{
        "satid": 45682,
        "satname": "STARLINK-1397",
        "intDesignator": "2020-035AB",
        "launchDate": "2020-06-04",
        "satlat": 51.7785,
        "satlng": 1.8116,
        "satalt": 552.2623
    }, {
        "satid": 46081,
        "satname": "STARLINK-1571",
        "intDesignator": "2020-055BG",
        "launchDate": "2020-08-07",
        "satlat": 52.3349,
        "satlng": 9.2484,
        "satalt": 552.5664
    }, {
```

Figure 3.7: Snippet of results from API: *https://api.n2yo.com/rest/v1/satellite/above/58.937/5.698/63.0/65/52apiKey=*. API key is needed to do this request.

From the response in figure 3.7 we can extract the *satlat*, *satlng*, and *satalt* from the satellites in view and calculate the distance from the dish. The *satid* can be used to identify which satellite is concerned. To calculate the distance from the satellites to a position on Earth we used Haversine formula [48]:

$$\begin{aligned} a = \; & \sin^2(\Delta latitude/2) + \cos(earthLatitude) \\ & * \cos(satelliteLatitude) * \sin^2(\Delta longtitude/2) \end{aligned} \tag{i}$$

$$c = 2 * a\tan 2(\sqrt{a}, \sqrt{1-a}) \tag{ii}$$

$$Distance = EarthRadius * c \tag{iii}$$

This equation takes in position in latitude, longitude, and same positions for the satellite. Then distance is calculated with the radius of the Earth. This formula is converted to Python code shown in the appendix A.11.

3.4.3 Experiment Automation

As the list of experiments gets longer, the need for automation increases. This can make the experiments much more efficient and time-saving. It also allows for repeatability, to make sure that experiments are always repeated under the same conditions.

3.4.3.1 Python Scripts

By using Python, automation of various tests can be done. The way this is set up for this thesis is through a Command Line Interface (CLI) script on the host controller from figure 3.2. When executing this script it first displays a set of options, these show the tests that are available. Then, depending on which option is chosen, it takes in some parameters that need to be set before the automated test can run. Operations like SSH, writing results to files, storing results in corresponding folders, executing tasks concurrently, logs to a log file, and so on. For lighter tests that run on the Raspberry Pi, single Python scripts on the device are executed.

3.4.3.2 Libraries

1. Pyshark

   Pyshark [23] is a wrapper for Tshark. It allows for packet parsing using Python. Pyshark was used to parse game capture PCAP files

2. Scapy

    Scapy [37] is a Python library for packet manipulation. In our case, we used Scapy to alter IP addresses and ports of packets in PCAP files

3. Matplotlib

    Matplotlib [45] is a Python library for visualization

4. Fabric

    Fabric [13] is a Python library used to do shell commands over SSH

5. Scipy

    Scipy [7] is a library for mathematical algorithms and equations built on the Python library Numpy [32]

3.4.3.3 File structure

This section seeks to provide a summary of the different folders and files for the thesis including diagrams and tables. In the tables, the code is referenced from the appendix for a more convenient overview.

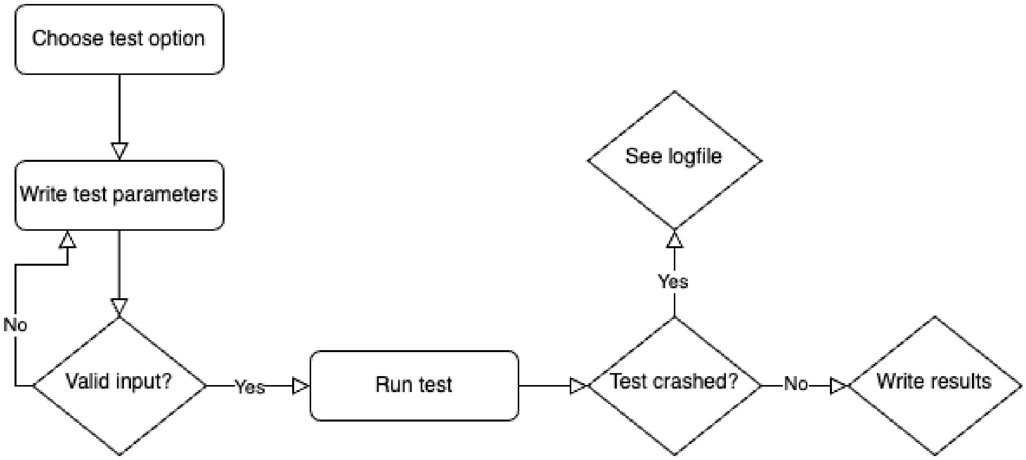


Figure 3.8: General flow of test automation

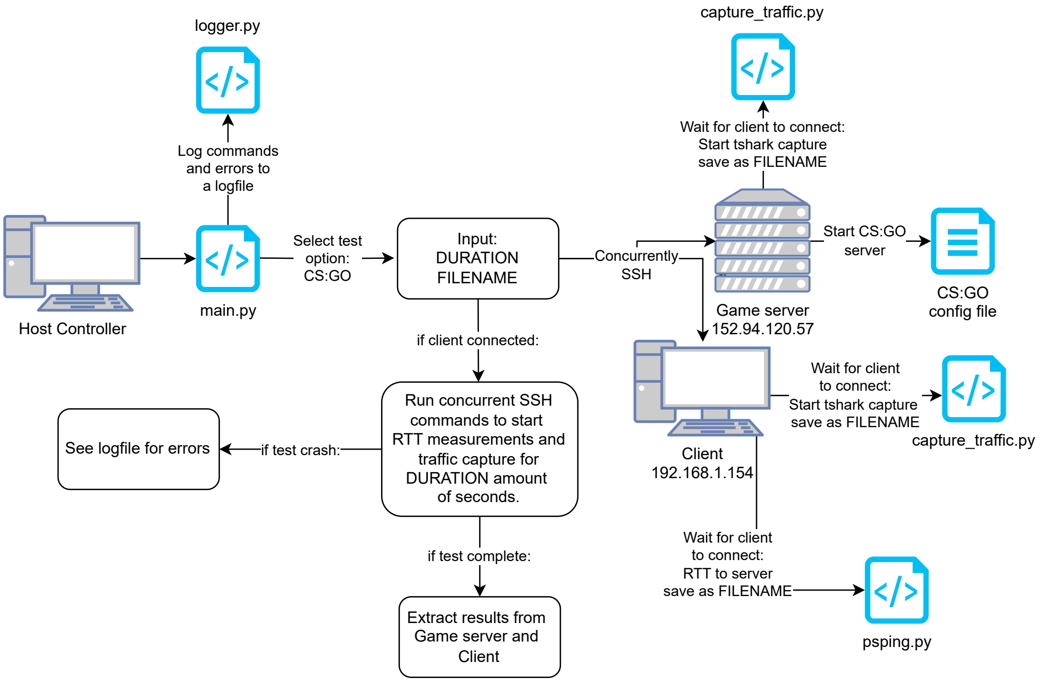


Figure 3.9: Flowchart of an automated test for a single CS:GO run

Figure 3.8 displays how the general flow of the test automation functions for the experiments. Figure 3.9 shows a more concrete example of how a run of CS:GO is tested. Figure 3.10 shows the file structure, containing folders with Python scripts and server configuration files.

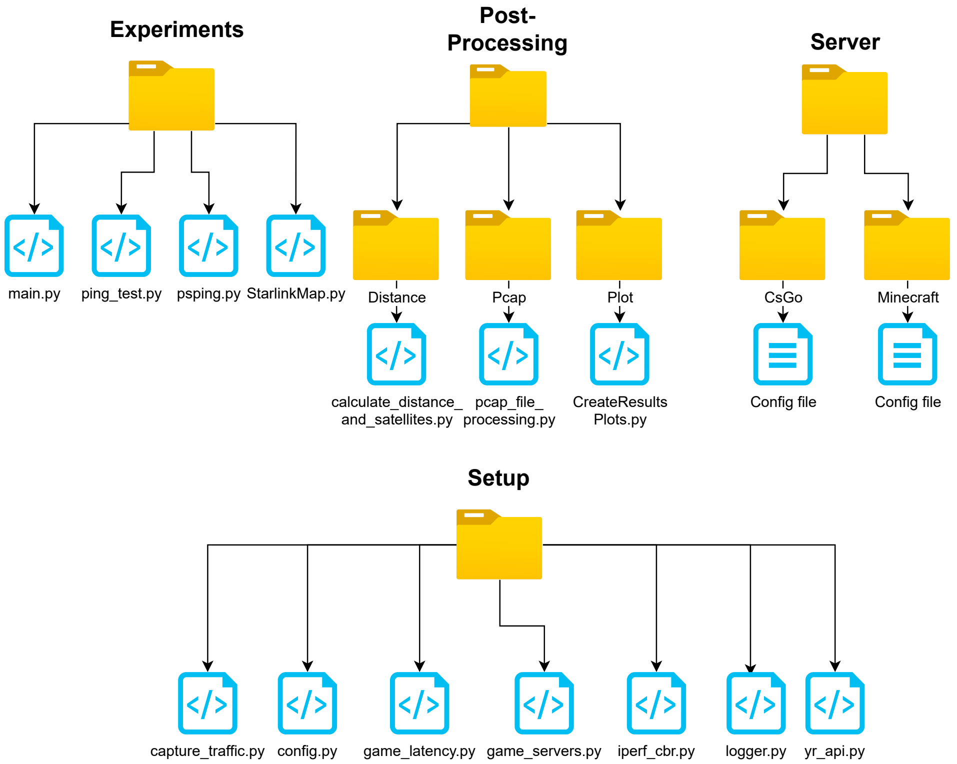


Figure 3.10: Diagram of the folders and files used in the thesis

Table 3.8 shows the files in the experiments folder 3.10, with a description of what the files do.

Reference	File name	Description
A.5	main.py	Where test automation code runs
A.6	ping_test.py	RTT measurement automation script
A.7	psping.py	RTT measurement using PsPing tool
A.8	StarlinkMap.py	Satellite API request

Table 3.8: Contents of the Experiment folder, files for running experiments

The table 3.9 shows the files used for post-processing located under the folder post-processing 3.10.

Reference	File name	Description
A.9	pcap_file_processing.py	Extracting and accessing PCAP files
A.11	calculate_distance_and_satellites.py	Script for calculating satellite distance
A.10	CreateResultsPlots.py	All plots for results

Table 3.9: Contents of the Post-processing folder, files to do post-processing of data and plotting

The setup table 3.10 contains Python files for setup before an experiment run.

Reference	File name	Description
A.12	capture_traffic.py	Capture traffic
A.13	config.py	Config for experimental scripts
A.14	game_latency.py	Sets up Ping script over ssh
A.15	game_servers.py	Starting game server over ssh
A.16	Iperf_cbr.py	Setup for running iPerf experiment
A.17	logger.py	Setup for logger
A.18	yr_api.py	Setup for yr api

Table 3.10: Contents of the Setup folder, files to setup the experiments

Server config files are listed in table 3.11, which shows the config files for the CS:GO and Minecraft server.

Reference	File name	Description
A.1	CS:GO DefaultProperties	Default properties for CS:GO server
A.2	CS:GO ServerProperties	Server Properties for CS:GO server
A.3	Minecraft DefaultProperties	Default properties for Minecraft server
A.4	Minecraft ServerProperties	Server properties for Minecraft server

Table 3.11: Contents of the Server folder, files to configure the game server

3.4.4 Post Processing

After the experiment automation process, the results need to be processed. The results vary from PCAP files to normal text files, so, therefore, a set of functions in Python is created to handle the different results. These functions do everything from pattern matching to generating plots. Most of the post-processing happens on a host controller, however, some of the results are processed on other machines.

3.4.4.1 Game Capture

All game traffic was captured and stored in PCAP files. A Python script (see A.9) was then written to extract certain data from different packet fields. All fields that can be accessed are found in the Pyshark GitHub repository [23].

The script starts by importing the Pyshark library:

```
import pyshark
```

A method in Pyshark called ”FileCapture” was then used to read the PCAP file and assign it to a variable. Next, there are two methods:

```
"get_tcp_data"
```

and

```
"get_udp_data"
```

that filters packets based on their transport layer field. The packets can then be filtered by source and destination IP addresses by using the method:

```
"get_pkts_by_ip"
```

The methods:

```
"list_of_pkts_per_second"
```

and

```
"list_of_bytes_per_second"
```

return lists of the number of packets and a number of bytes sent per second respectively. The main section of the script in A.9 is an example of processing a CS:GO game and generating lists of bytes per second sent between the server and the client.

3.4.4.2 Text files

Processed data was written into text files to make analysis and visualization easier. For RTT measurements and Constant Bit Rate (CBR) tests, RTT and throughput values were parsed and written to text files. As for packet and byte rate values, once calculated, were written to text files. All text files were structured in a way where there is one value for each line in the file.

3.4.4.3 Synthetic Packet Pairs

Synthetic Packet Pairs (SPP) is a CLI tool developed by the Centre for Advanced Internet Architecture [2] to measure RTT based on network data captured at the sender and receiver end, without the need for clock synchronization. It provides passive RTT measurement which means that it does not require running simultaneously to the network capture. This tool works on Linux and FreeBSD based and is installed on a separate machine running FreeBSD.

The way that it works: SPP uses two measurement points or MP's, MP$_{ref}$ (reference) and MP$_{mon}$ (monitor). Both MP's capture the network traffic of interest(sent and received), for example using Tshark. SPP takes in two PCAP files as input, one for each MP. It also needs the corresponding IP address for each of the MPs. For every recorded packet both MP$_{ref}$ and MP$_{mon}$ log a timestamp (ts) representing when the packet was captured, and a short 'Packet ID' (PID). The PID is calculated from a hash function (e.g. CRC32) over key bytes within the packet. SPP creates two lists for each of the MPs. One packet within the list consists of a PID and a timestamp. The two lists are then combined to identify packet pairs and calculate RTT.

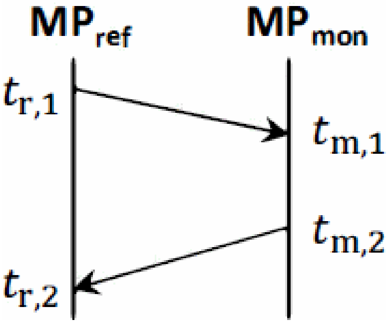


Figure 3.11: SPP packet pairing algorithm

To better explain the packet pair algorithm a short-hand notation needs to be defined. $t_{j,i}$ where t represents the timestamp, i represents MP$_{ref}$ (r) or MP$_{mon}$ (m), and j represents the first (1) or second (2) packet of a packet pair. In figure 3.11 we see an example of how the algorithm works. SPP assumes that a packet is used in at most one packet pair. It also searches for the closest packet pair where $t_{m,2}$ > $t_{m,1}$. Once a packet pair has been discovered, the RTT calculation is straightforward, as shown in figure 3.12 [52].

$$RTT = (t_{r,2} - t_{r,1} - (t_{m,2} - t_{m,1}))$$

Figure 3.12: SPP RTT calculation equation

3.4.4.4 Visualization

To visualize the results gathered from the experiments the Matplotlib Python library was used as described in 3.4.3.2. We used Matplotlib to create quality plots from data collected after each experiment.

To begin using Matplotlib in Python, start by including this line in the Python file:

```
import Matplotlib.pyplot as plot
```

This command will import the pyplot library from Matplotlib and make it usable in the code.

For some of the data collected, the need for smoothness was apparent. To smooth out data, we used a Python library called Scipy [7] 3.4.3.2 and imported it like this:

```
import Scipy.signal.savgol_filter as sc
```

This code line imports the Savitzky-Golay filter used to smooth out data. This filter takes in 3 important parameters: Data (array), window length, and poly order.

3.4.5 Other Setup Configurations

3.4.5.1 Security

Because the server hosts games, the network has to open ports to let traffic in and out. This leaves a vulnerability, the opened ports can be exploited if not handled correctly. Underneath is a list of security measures that are applied:

1. SSH keys are generated for the machines that use SSH to access the server
2. SSH as the root user is disabled
3. SHH using a password is disabled (only SSH keys accepted)
4. Default SSH port (22) is closed
5. Firewalld installed and blocks traffic from all ports except the ones used for the games
6. A Crowdsec bouncer is installed to block unwanted IP addresses that try to access through the opened ports

3.4.5.2 Difficulties

During this thesis, we encountered difficulties. Some related to one of the tools that were used, and others to the network. The difficulties and how they were solved are listed underneath:

1. The SPP tool was not compatible with Debian 11. To be able to use the SPP tool it was installed on another computer running FreeBSD (not the server

in figure 3.2 as initially intended). FreeBSD contrary to Debian 11 is compatible with SPP. This resulted in moving the files that needed to be processed by SPP to the FreeBSD computer, and then copying them back after they have been processed.

2. Another problem occurred when SPP for the first time processed the captured traffic and outputted nothing. Starlink is operating with Network Address Translation, which means that IP addresses and ports are logically changed between the endpoints. This results in SPP not recognizing the IP addresses and ports for the different endpoints in the two PCAP files, hence not returning any output.

   To solve this problem Scapy from 3.4.3.2 is used. With this library, we can manipulate IP addresses and ports in the PCAP files so that they match, this way SPP can successfully measure packet pairs.

3. ICMP packets are blocked into the UiS network where the game server is connected. This created some complications regarding RTT measurements for the server. To solve this issue TCP ping was used instead. This allowed for RTT measurement with TCP packets instead of ICMP.

4. We experienced difficulties with measuring packet loss in the Minecraft game data. With TCP as the transport protocol, the TCP agent will retransmit lost segments. TCP packet loss can thereby be interpreted as an increase in latency [35]. Given this and this project's time restraints, we decided to prioritize latency and throughput measurements of Minecraft game traffic.

Chapter 4

Experiments and Results

To conduct this research, a set of scenarios have been created. Most of the scenarios contain sub-scenarios for a more in-depth study. The results are mainly divided into two parts, one part for baseline measurements, and one for online gaming measurements. The table 4.1 shows a brief overview of the different sub-scenarios.

Scenarios	Description
4.1.1	Baseline latency
4.1.2	Available satellites
4.1.3	CBR measurements
4.1.4	VBR measurements
4.2.1	CS:GO single run
4.2.2	CS:GO all runs
4.2.3	Minecraft single run
4.2.4	Minecraft all runs

Table 4.1: Summary of sub scenarios

4.1 Scenario 1 - Baseline measurements

Scenario 1 will run baseline measurements of the Starlink broadband service. This includes RTT, constant bit rate tests, variable bit rate tests with different congestion control algorithms, and throughput. This will indicate how Starlink performs. For each of these experiments, the client device will always be at UiS directly connected to the Starlink router.

4.1.1 Ping Starlink Gateway

Latency can play a huge part when it comes to online gaming. This experiment seeks to find the RTT from a host to the first hop back on earth from the Starlink satellites. See figure 3.1 for the topology. The host for this experiment will be a Raspberry Pi. Traceroute will be used to find the Starlink gateway IP address, and then the host uses ping to send an ICMP echo request to this IP. This way we get an estimate of RTT over the Starlink satellites. The experiment runs for 24 hours sending bursts of 1000 ICMP echo requests with one-second intervals, every 15 minutes. This process will be automated through a Python script.

4.1.2 Overview over available satellites

Distance to the satellite can play a big role in round trip time, and when hands-on/off satellite switching is happening. To get a better understanding of what is happening, this experiment investigates the correlation between the distance of the closest satellites and RTT.

As per 4.1.1, we used Starlink gateway as the address used as the receiver for Ping. This experiment runs for 500 seconds with an ICMP request every second. Parallel to the Ping running, to get an overview of what satellites are available each second, an API 3.4.2.4 request is sent. This API returns a JSON 3.7 of every satellite in view, with information on latitude, longitude, and altitude. From this information, the distance of the closest satellite is calculated in Python.

The Python script A.8 shows the process of running the API request and Ping concurrently, and the Python script A.11 calculates the results afterward.

4.1.3 Constant Bit Rate

This experiment aims to measure the packet loss of the Starlink service when there is constant traffic. This will be done through a 24-hour-long CBR test. By utilizing iPerf, the server from figure 3.2 will send UDP traffic with a constant bit rate of 10, 20, and 30 Mbps. Two times an hour, within the 24-hour window, a

5-minute run is carried out for each bit rate value. Therefore, each hour is represented with two runs of 5 minutes with a 30-minute interval in between, this counts for all bit rate values. See figure 4.1 for an example of a one-hour window. The cloud coverage, during the test, will also be recorded and displayed adjacent to the packet loss. This experiment gets automated through a Python script on the Raspberry Pi.

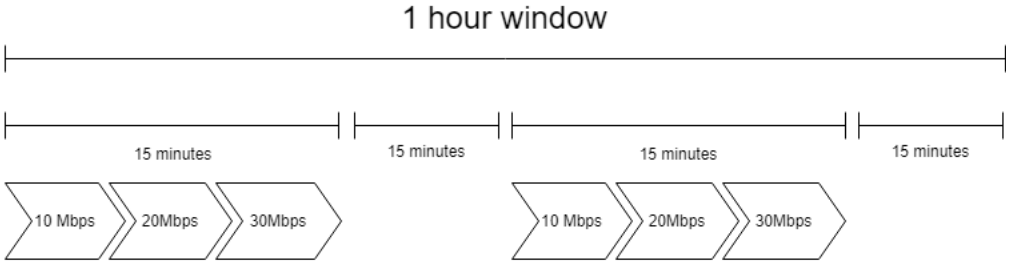


Figure 4.1: One hour window of how CBR experiments execute

4.1.4 Variable Bit Rate

Unlike the CBR test, this Variable Bit Rate (VBR) test aims to measure throughput over Starlink with different congestion control algorithms, see figure 4.8. The algorithms that will be used are Reno, Cubic, Bottleneck Bandwidth, and Round-trip propagation time (BBR). Again by utilizing iPerf the server from figure 3.2 will send traffic, but this time it is going to use the TCP protocol, not UDP. The experiments will run for 60 seconds for each congestion control algorithm.

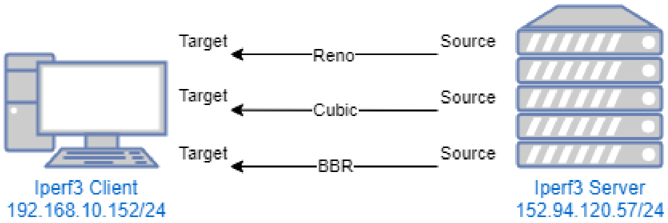


Figure 4.2: iPerf server sending TCP traffic using different congestion control algorithms

4.2 Scenario 2 - Gaming measurements

Scenario 2 aims to measure the gaming QoS on two different games. The games that will be played are CS:GO and Minecraft. Each game will have ten runs consisting of five minutes of game-play. This results in four sub-scenarios, two for a closer look at a single run for both games and two for a wider look at all the runs for both games. All the runs are going to be played on the client PC, and hosted on the game server in figure 3.2. Traffic will be captured along with latency measurements.

4.2.1 CS:GO Single Run

This sub-scenario will take a closer look at a single five-minute run of CS:GO over Starlink. Metrics such as RTT, throughput, and packet loss will be measured. To have some comparison, a similar five-minute run will be carried out over the terrestrial internet. Since the CS:GO game traffic is UDP, and to be able to measure RTT and packet loss, capturing of the packets needs to be done on both the client and server end. For this task, Tshark is going to be used. To ensure that the traffic capture starts and stops at the same time, this process will be automated using SSH from a host controller in a Python script, see figure 3.2. For RTT measurements two different tools will be utilized, TCP ping and SPP. These are also going to be automated using Python.

4.2.2 CS:GO All Runs

This experiment is similar to the CS:GO single-run scenario with regards to metrics and automation, however, it will look at all ten runs. It will look at averages over the different runs. This is to get a general understanding of how the traffic behaves over multiple runs and whether Starlink is stable enough or not.

4.2.3 Minecraft Single Run

Like CS:GO single run, this experiment will also look at a single five-minute run of Minecraft over Starlink. The metrics to be measured are RTT and throughput. TCP ping and Tshark will be used for measuring RTT, and Tshark for game traffic capture. This process will be automated using Python scripts along with SSH from a host controller, see figure 3.2. A comparison between Starlink and terrestrial RTT will also be added.

4.2.4 Minecraft All Runs

This experiment will also look at all ten runs, only for Minecraft. it will look at averages over the different runs, and go through the same metrics as for Minecraft single run. This is to give an idea that Starlink is capable of playing Minecraft.

4.3 Results

This section presents the results for the different scenarios. They are represented as plots with some describing text on what we see.

4.3.1 Ping Starlink Gateway Results

Figure 4.3 is a single run gathered from all of the runs which are displayed in figure 4.4. One ICMP echo request is sent every second for 1000 seconds. The test started at 16:00 Monday 6. march 2023 and ended at 06:45 Tuesday 7. march 2023.

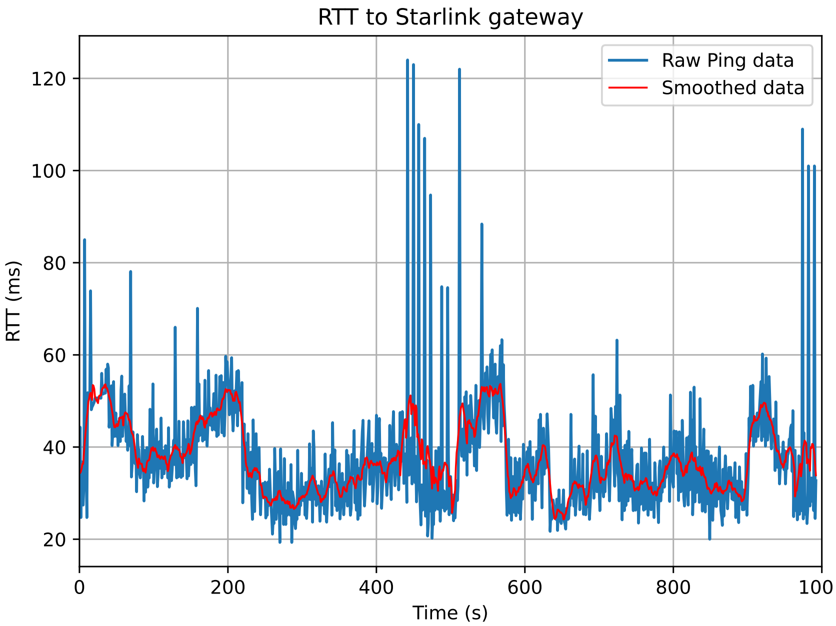


Figure 4.3: RTT measurement to Starlink gateway with one echo request each second for 1000 seconds. The smoothing filter used is Scipy´s Savgol filter with a window length of 21

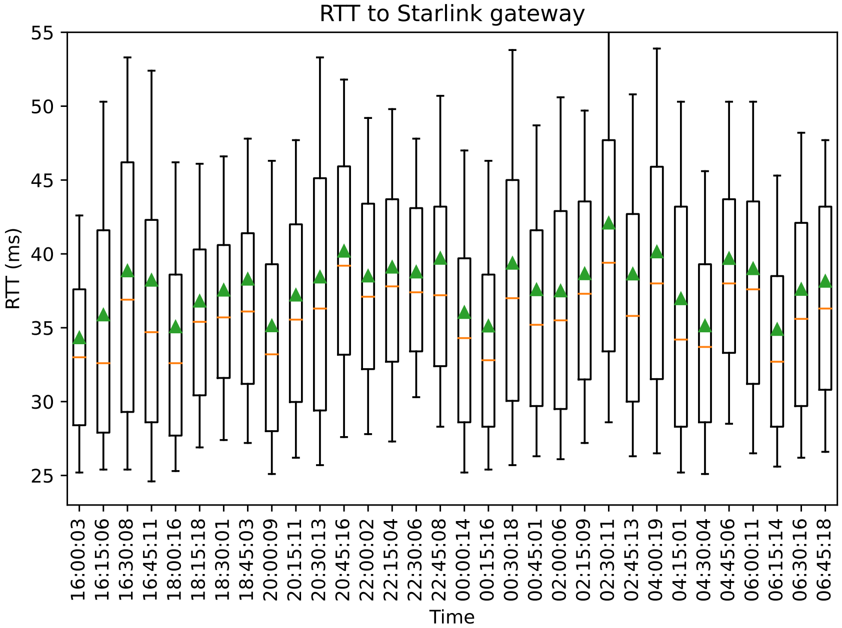


Figure 4.4: Ping test to Starlink gateway. The boxes are configured with 10th and 90th percentiles. Each box contains a measurement of one ICMP request every second for 1000 seconds

From the collective data 4.4 we see the trends of the orange line (median) and green arrowhead (mean) is contained in a relatively small window of around 10ms. The data was captured from Monday 6. to Tuesday 7. March 2023. The weather on this run was stable, with no clouds and no precipitation.

4.3.2 Overview over available satellites results

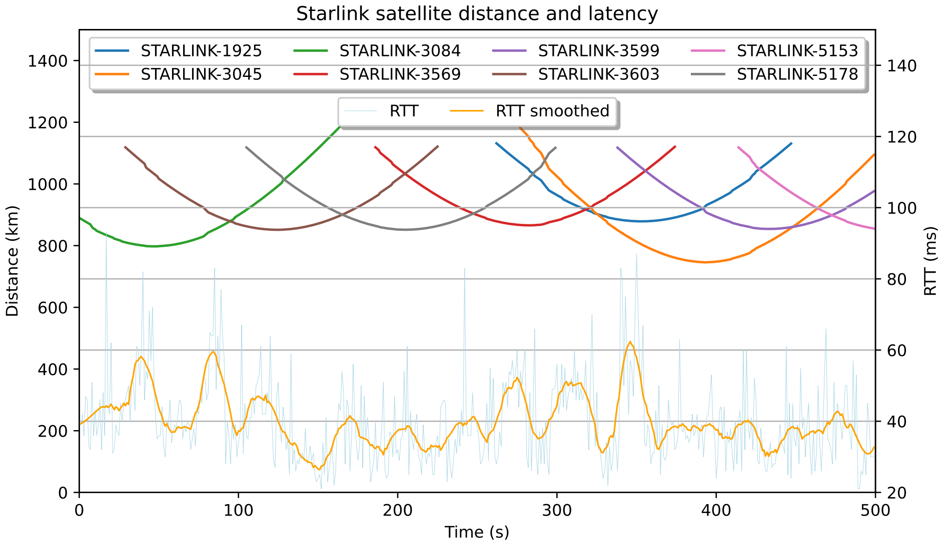
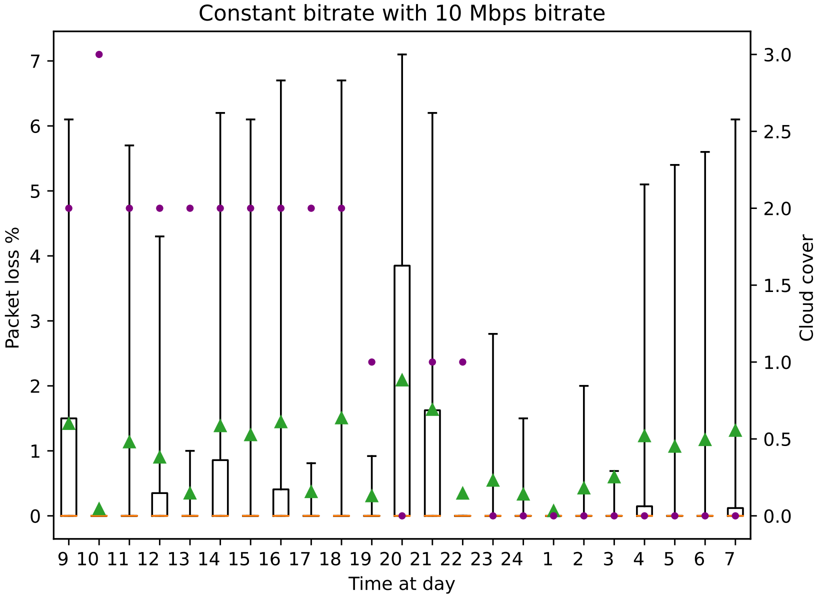


Figure 4.5: Satellite distance from Starlink dish location 3.3.4, together with result from ping to Starlink gateway (100.64.0.1). The smoothing filter used is Scipy´s Savgol filter with a window length of 30 on the RTT

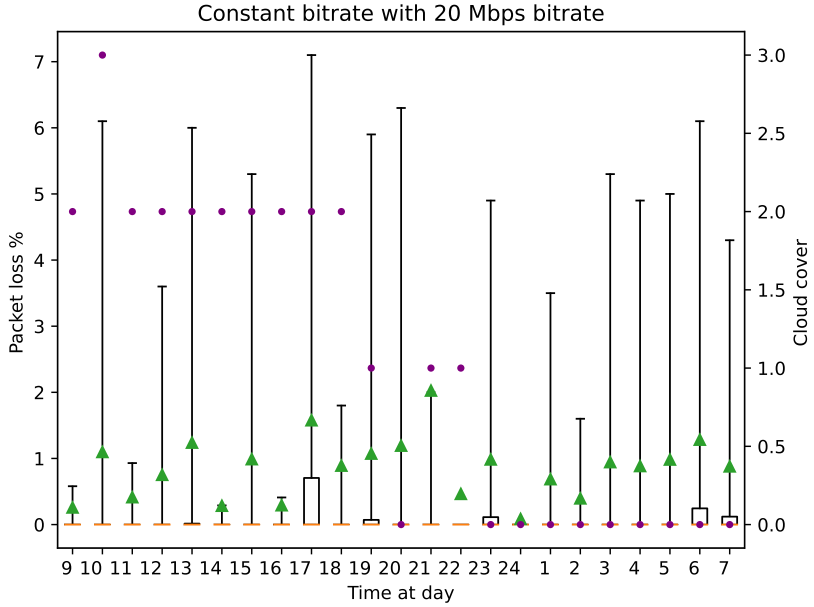
Figure 4.5 shows the satellites in view at the same time as the latency experiment. This is a 500 seconds run of RTT measurements to a Starlink gateway and API request to gather satellite positional data. For the RTT graph, a smoothed line is added with a filter 3.4.4.4 to aid visualization. The figure shows satellites coming in/out of view at given times, with overlapping distances from the ground. The satellites stay in view up to a distance of around 1200 km. In the time interval of 500 seconds, we see eight different satellites in view.

4.3.3 Constant Bit Rate Results

In figure 4.6 and 4.7 the packet loss median stays consistently at zero percent, however, the mean varies a bit around one percent. The runs were measured at 05.April 2023.



(a) CBR 10 Mbps



(b) CBR 20 Mbps

Figure 4.6: Constant bitrate baseline run with 10 Mbit/s and 20 Mbit/s, 2nd y-axis shows the cloud cover value from each run. Each run is a 5-minute test and started at 09.00 05.04.2023. The boxes are configured with 10th and 90th percentiles. The purple dot is the cloud cover value, and the green is the average measurement

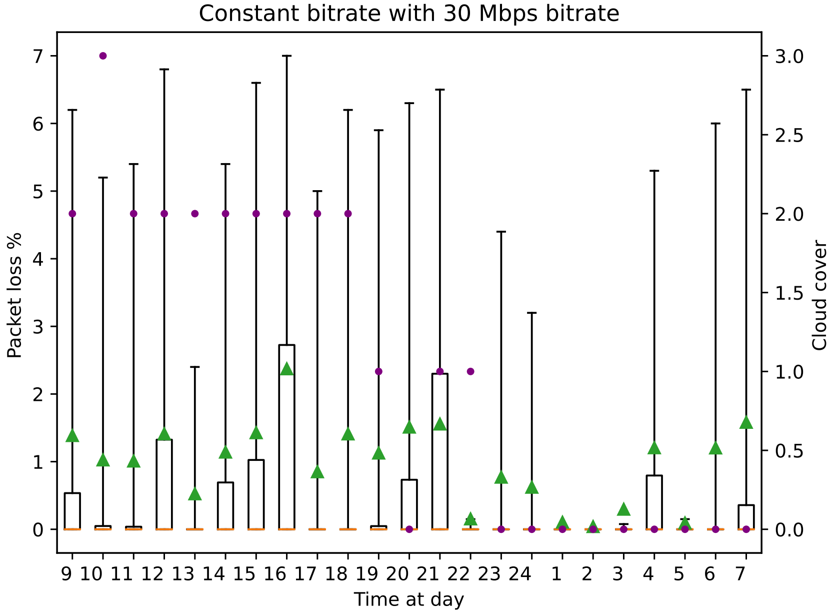


Figure 4.7: Constant bitrate baseline run with 30 Mbit/s, 2nd y-axis shows the cloud cover value from each run. Each run is a 5-minute test and started at 09.00 05.04.2023. The boxes are configured with 10th and 90th percentiles. The purple dot is the cloud cover value, and the green is the average measurement

The figures 4.6and 4.7 shows experiments described in scenario 4.1.3. These figures display a constant bit rate test with fixed bandwidth, packet loss in percent, and cloud coverage on the y-axis.

4.3.4 Variable Bit Rate Results

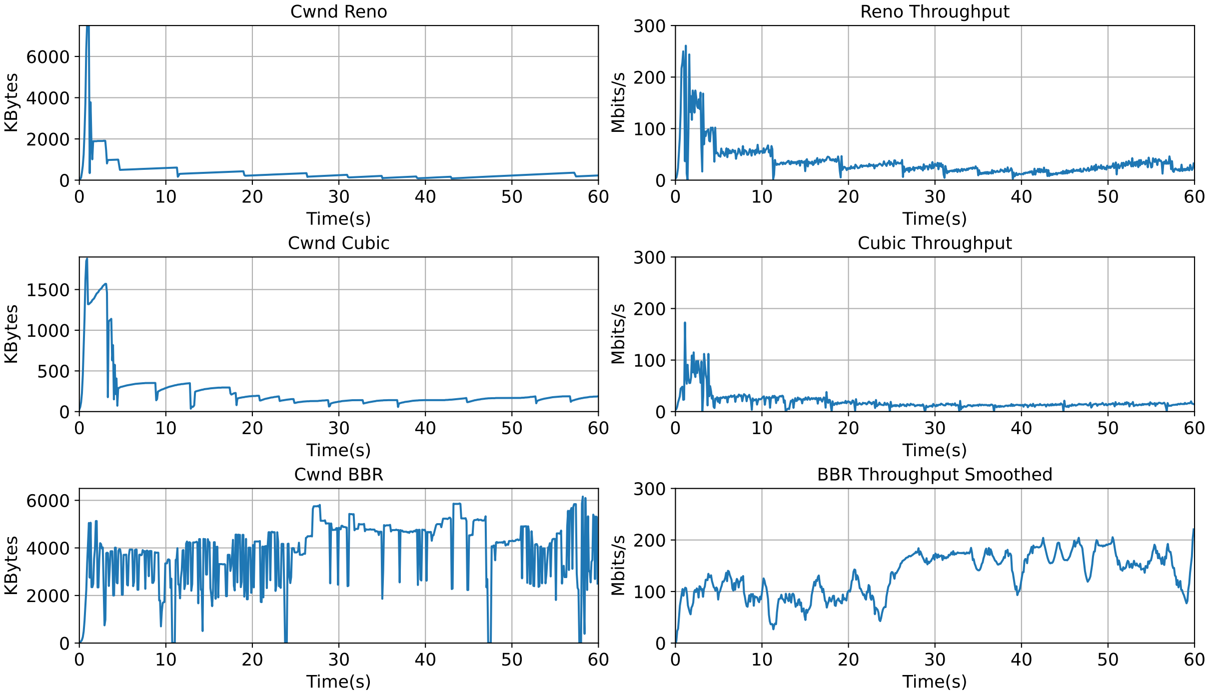


Figure 4.8: VBR downlink test with Reno, Cubic, and BBR, for 60 seconds each with 0.1-second interval. BBR throughput is smoothed with a window length of 18. The Congestion Window (cwnd) is on the server endpoint, and the throughput is measured on the receiving end

Figure 4.8 shows the results of experiment 4.1.4 with a variable bit rate. The figure displays the usage of three different congestion controls: Reno, Cubic, and BBR. Cwnd and throughput results of each Congestion Control Algorithm (CCA) are plotted against each other. Reno's throughput gradually steps down until it stabilizes at around 35 Mbit/s. The same can be observed for Cubic until it reaches a throughput of approximately 20 Mbits/s. BBR, however, achieves an average throughput of 131 Mbit/s.

4.3.5 CS:GO Single Run Results

The results for this run are taken from run number 9. See figure 4.14.

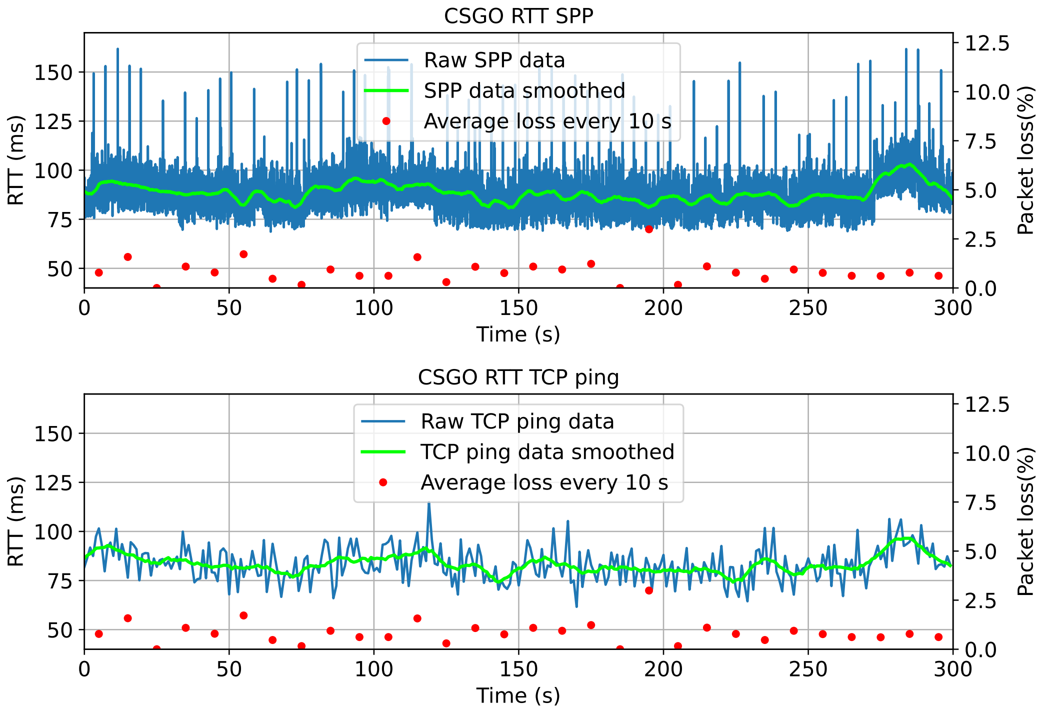


Figure 4.9: RTT measurements over Starlink during a CS:GO run using SPP and PsPing (TCP ping). Smoothed using Scipy's Savgol filter with a window length of respectively 1000 and 22. This gives a window length factor of around 15

In figure 4.9 and 4.10 we observe that RTT measurements for SPP has more data points (around 15 000) than that of TCP ping. TCP ping measures RTT once every second which equals 300 data points in both figures. The RTT over Starlink with SPP ranges between 70-120ms with spikes up to 150ms, and the RTT with TCP ping ranges between 70-110ms.

For the terrestrial run in figure 4.10 the RTT with SPP ranges between 3-6ms with spikes up to 15ms, whereas the RTT with TCP ping ranges between 4-7ms.

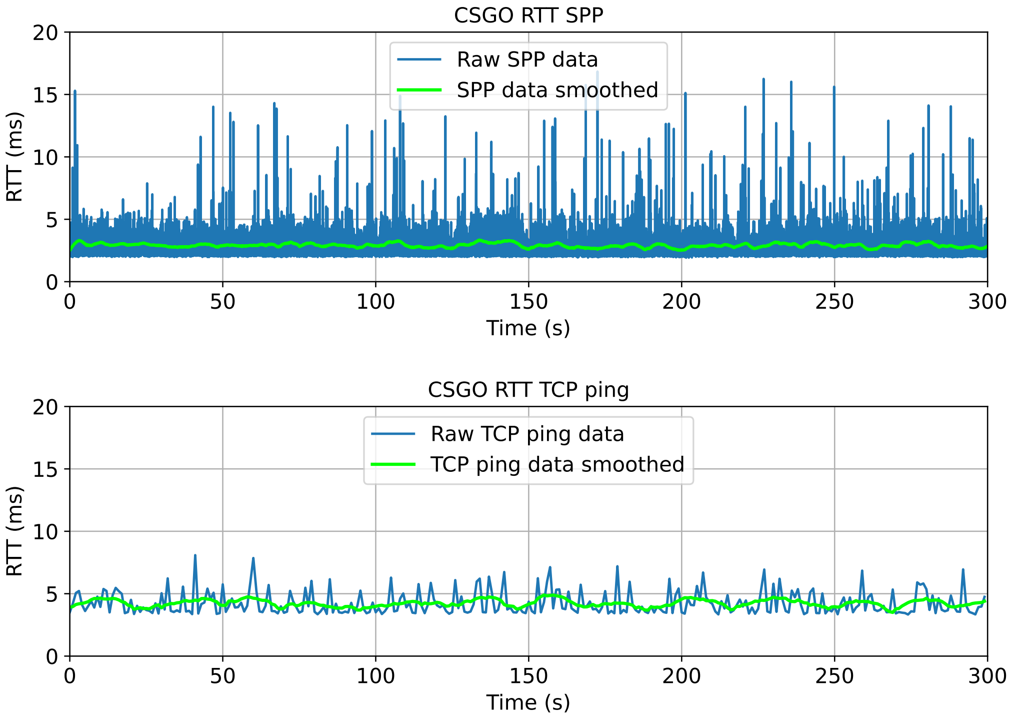


Figure 4.10: RTT measurements over terrestrial internet during a CS:GO run using SPP and PsPing (TCP ping). Same smoothing method and window lengths as in figure 4.9

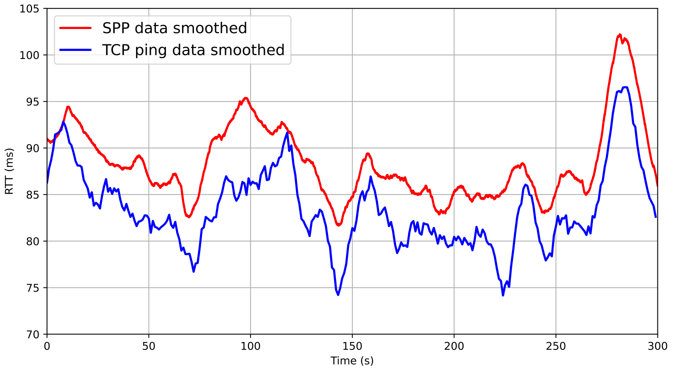


Figure 4.11: Smoothed RTT data from SPP and TCP ping in figure 4.9

In figure 4.11 we see that the smoothed results from TCP ping and SPP follow a similar trend, however, the TCP ping data averages a bit lower than the SPP data. Note that SPP is measuring RTT on the actual game traffic (passive measurement), whereas TCP ping is an active measurement that sits on top of the game traffic.

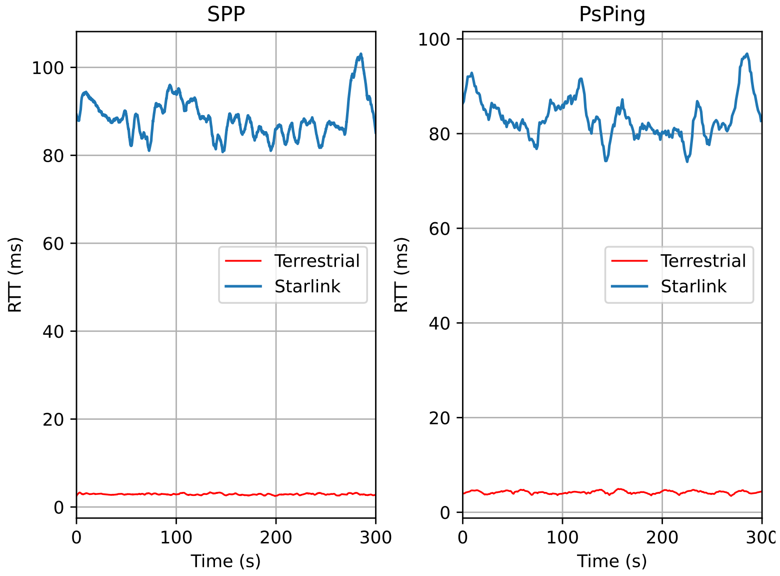


Figure 4.12: Comparison of RTT (measured with PsPing (TCP ping) and SPP) in CS:GO between Starlink and terrestrial

Figure 4.12 is showing a comparison of RTT in terrestrial and Starlink networks, on a CS:GO gaming run. The terrestrial RTT is much more stable and consistent than the Starlink network. The difference in RTT on this run is in general around 70ms.

In figure 4.13 we see a five-minute run of CS:GO on Starlink and over terrestrial internet. The throughput on the two runs differs with 61 Kbit/s. On the terrestrial link, there is virtually no packet loss compared to Starlink that have around 1% on average. The RTT (measured with TCP ping) on Starlink stays regularly between 70-100ms, and for the terrestrial run, it stays between 4-6ms.

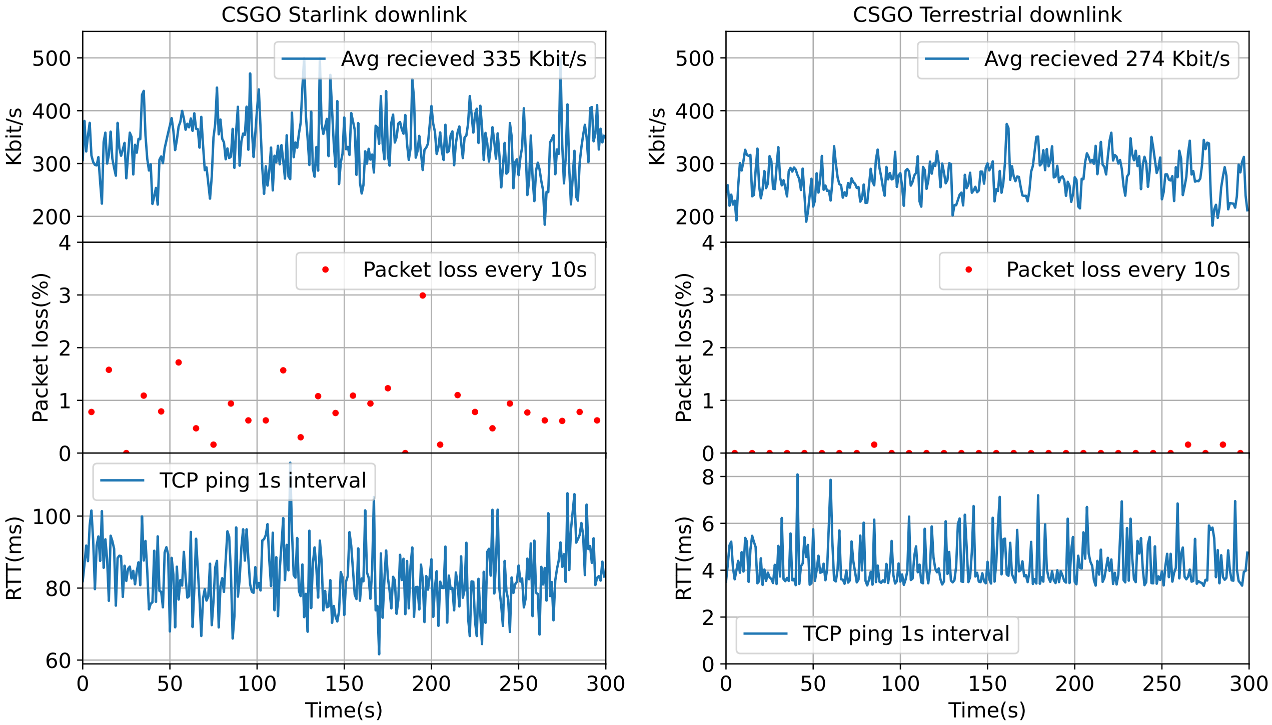


Figure 4.13: Downlink throughput, packet loss, and RTT in a single CS:GO run for Starlink and terrestrial internet. PsPing (TCP ping) is used to measure RTT

4.3.6 CS:GO All Runs Results

The CS:GO game data was captured on March 27th, consisting of ten five-minute captures. The mean is represented as arrowheads and the median is a line.

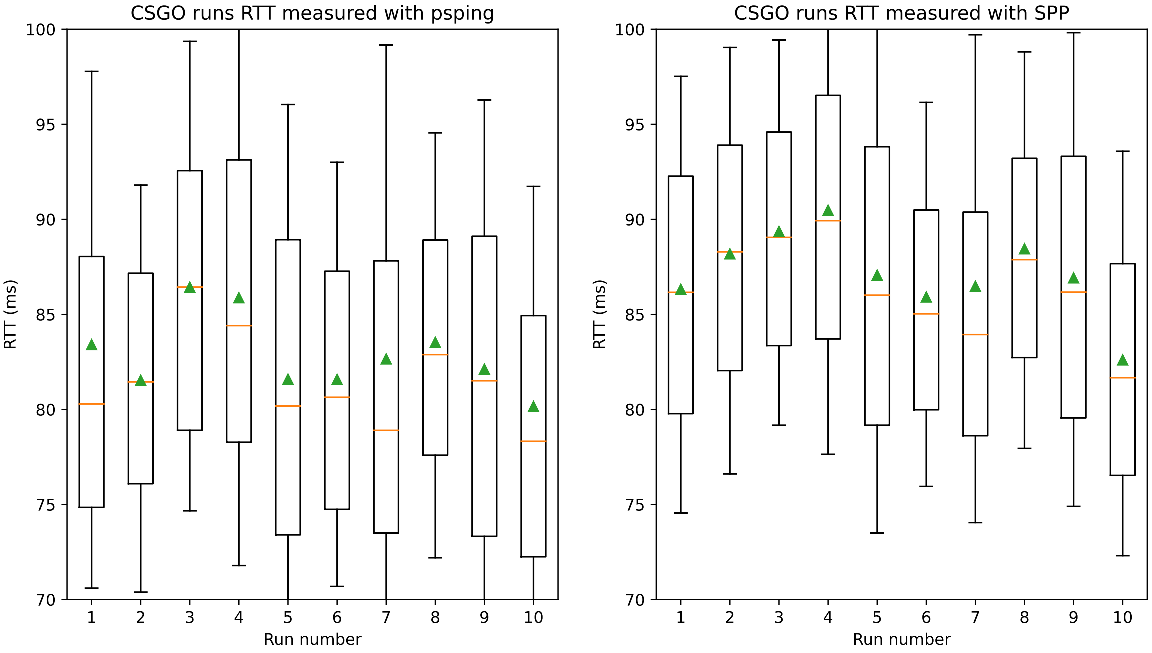


Figure 4.14: PsPing (TCP ping) and SPP is used to measure RTT for all CS:GO runs. Distribution edges are set to 10 and 90 percentile

In figure 4.14 it is evident that the two box plots share a similar pattern. Here we also notice that the RTT data from SPP is slightly higher than that of TCP ping. SPP results in a median of around 83ms, and for TCP ping it is slightly lower with a median of around 80ms.

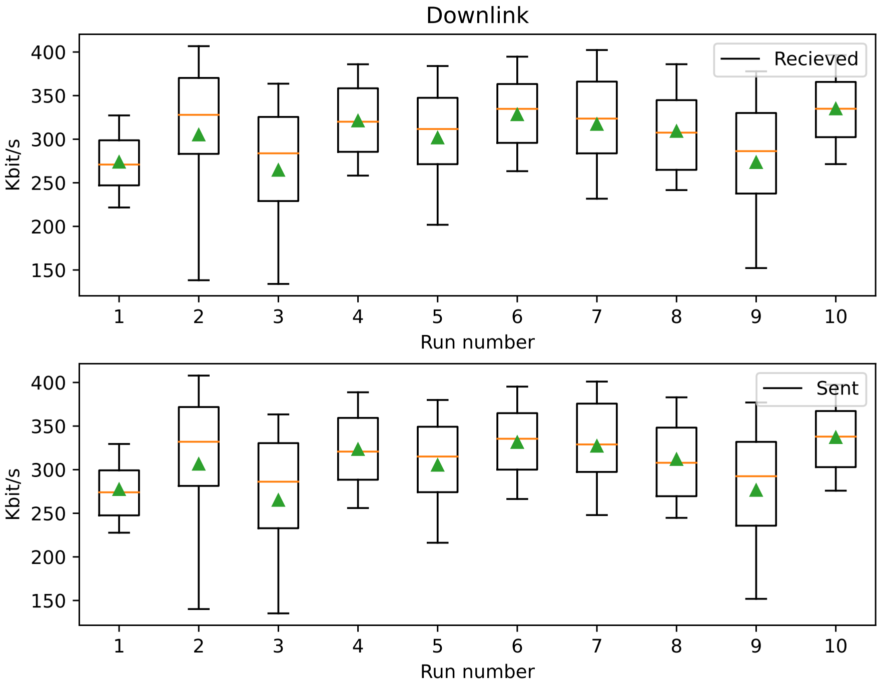


Figure 4.15: CS:GO downlink Bitrate for ten runs with five minutes each, the boxes are configured with 10th and 90th percentiles

Figure 4.15 shows the traffic sent and received on the downlink. We see only a negligible difference between the sent and the received traffic. This coincides with our findings for packet loss in 4.13 a single run, where the loss average is about 1%. The difference between the sent and received traffic seems to be consistent for all the respective runs.

4.4 Minecraft Single Run Results

The results for the Minecraft data are from run number six in figure 4.20. All smoothing methods of graphs use Scipy's Savgol filter from the Scipy library [7].

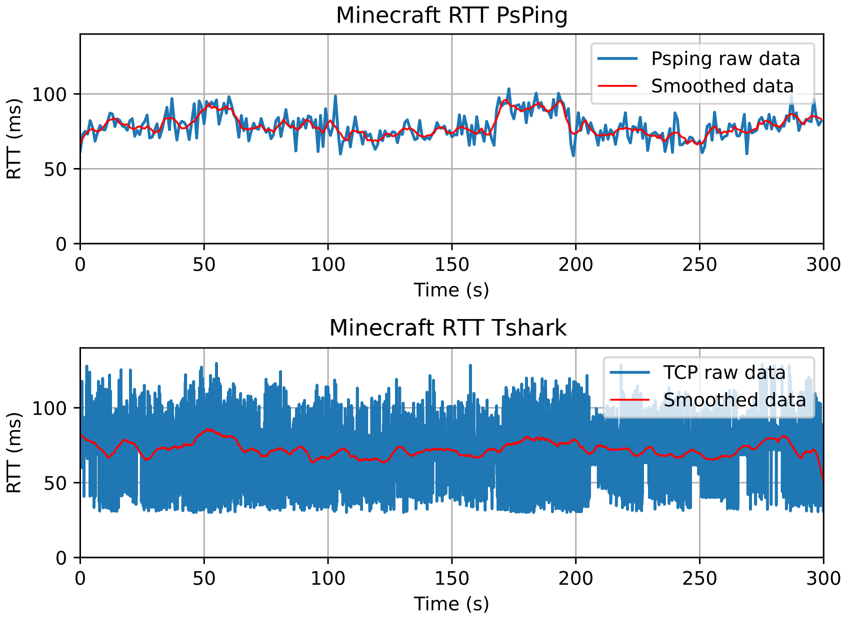


Figure 4.16: RTT measured with PsPing (TCP ping) and with Tshark from the TCP packets. Grapsh are smoothed using Scipy's Savgol filter with a window length of 20 and 1000

From figure 4.16 note that TCP ping measures the RTT once every second meanwhile the RTT of the TCP packets is measured for every packet sent in the game capture (in this case containing 13436 data points). TCP ping measurements show RTTs in a range of about 55ms on the lower end up to about 100ms on the higher end. In the second graph, RTTs vary greater, with latency somewhere between about 30ms and about 125ms. The median RTT is 75.55ms for the TCP packets.

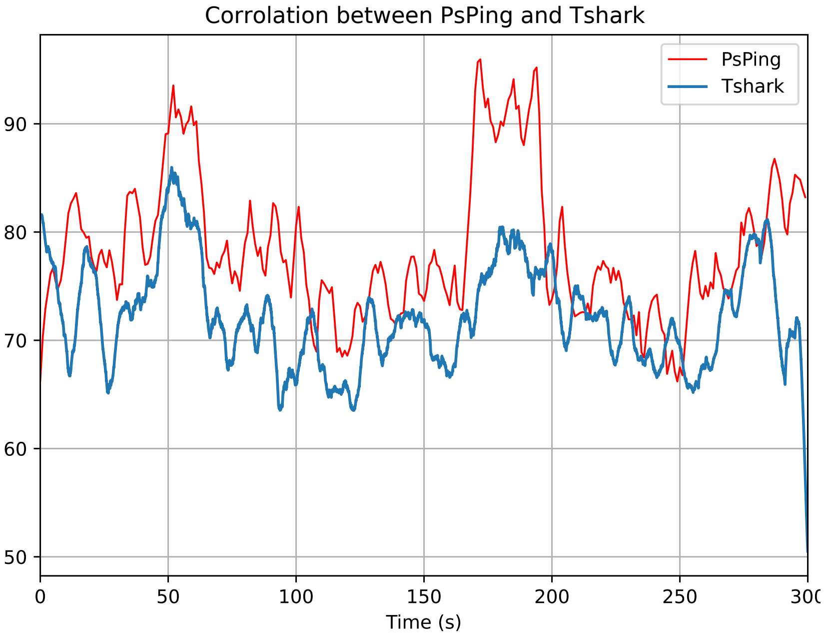


Figure 4.17: Corrolation plot between smoothed RTT measurement with PsPing (TCP ping) and Tshark from TCP packets. Graphs are smoothed using Scipy's Savgol filter with a window length of respectively 20 and 1000.

Figure 4.17 is showing smoothed RTT from TCP packets and PsPing (TCP ping) together. The graphs seem to have the same trends, although the TCP ping data averages a slightly higher latency, the two share similar sudden ups and downs in latency over the five minutes.

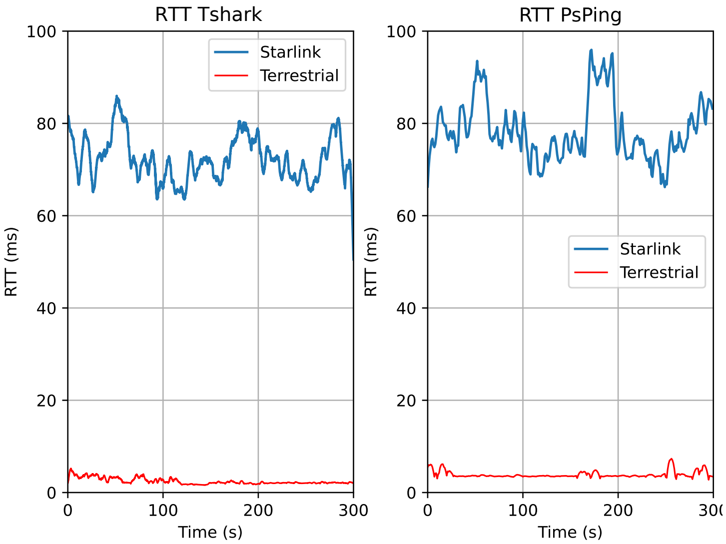


Figure 4.18: Comparison of RTT in Minecraft between Starlink and terrestrial internet with measurement tools: Tshark and PsPing (TCP ping)

Figure 4.18 is showing a comparison of RTT in terrestrial and Starlink networks, on a Minecraft gaming run. The terrestrial RTT is much more stable and consistent than the Starlink network. The difference in RTT on this run is in general around 70ms.

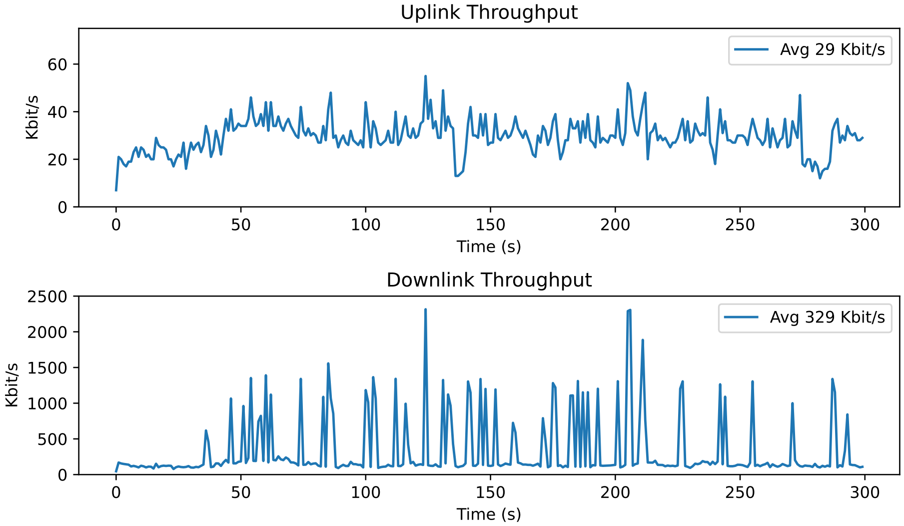


Figure 4.19: Minecraft uplink and downlink on run number 6 of Minecraft gaming

The throughput on the links in figure 4.19 are asymmetrical. On the uplink, the throughput lies mostly between 25 Kbit/s per second and 50 Kbit/s per second with some exceptions. As for the downlink, the throughput is mostly in the range of 10 Kbit/s per second to 20 Kbit/s, but it experiences significant traffic bursts of up to about 2300 Kbit/s per second. Why this could be the case will be addressed in the discussion section.

4.4.1 Minecraft All Runs Results

The Minecraft game data was captured on March 28th, consisting of ten five-minute captures. In this section, our findings are represented in box plots. For the plots, the distribution edges are set to 10 and 90, the mean is represented as arrowheads and the median is displayed as a line.

Figure 4.20 shows RTT values using TCP ping and TCP packet measurements for all the Minecraft game captures.

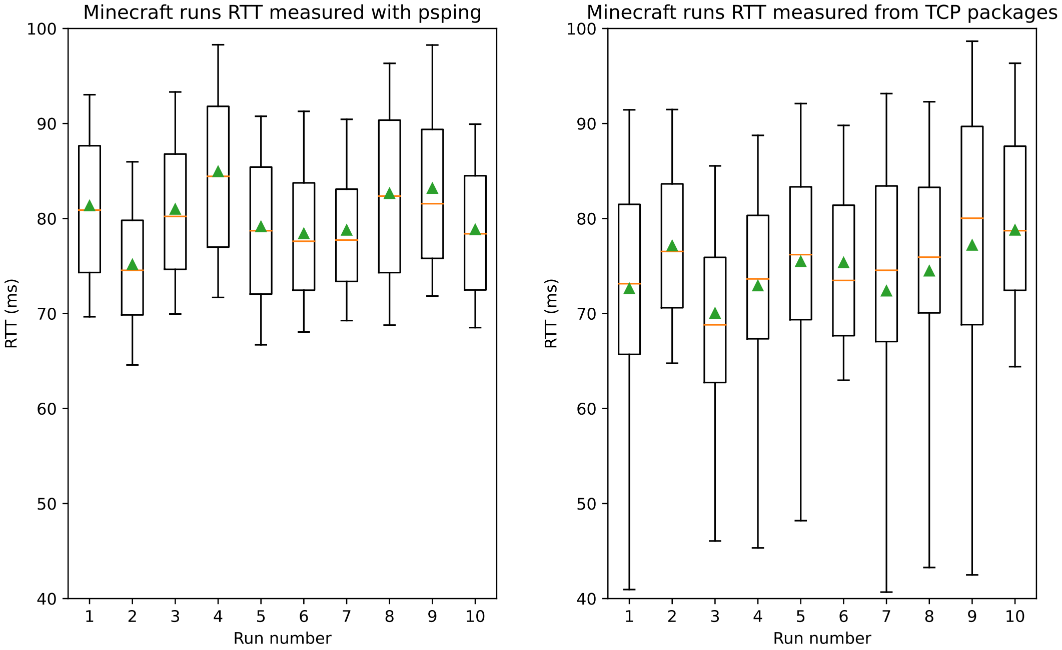


Figure 4.20: RTT data from 10 Minecraft gaming runs, measured with PsPing (TCP ping) and on TCP packets with Tshark. The boxes are configured with 10th and 90th percentiles

Results show consistent RTT distributions with TCP ping measurements and with TCP measurements but to a lesser extent. Overall, neither of the plots appears to be skewed in any direction. With TCP ping the median RTT ranges from 75ms to 85ms. As for the captured TCP data, median RTTs are slightly lower in a range from 68ms to 79ms. Given the significantly larger data set, the second box plot indicates a greater variance in RTT. Each box in the first plot consists of 300 RTT values, whereas in the second plot, the boxes consist of about 13 thousand to 14 thousand RTT values.

Figures 4.21 and 4.22 show Kbits per second for all runs on the uplink and the downlink respectively.

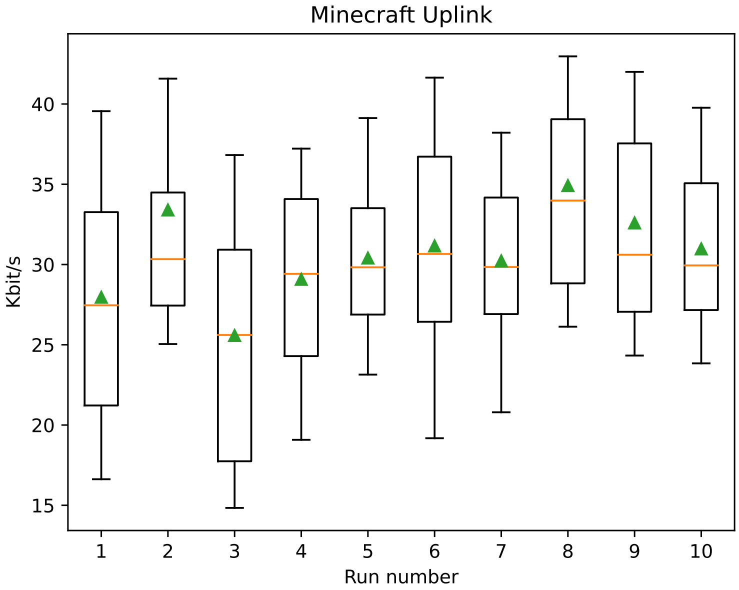


Figure 4.21: Boxplot with Kbit/s on the Uplink for 10 runs of gaming in Minecraft. The boxes are configured with 10th and 90th percentiles

We see that throughput varies on the uplink. For instance, the difference in average Kbit per second between the second and the third run is close to 8 Kbit/s. The median figure is in a range between 26 Kbit/s and 34 Kbit/s. It appears that in half of the test runs the box plots are skewed upwards.

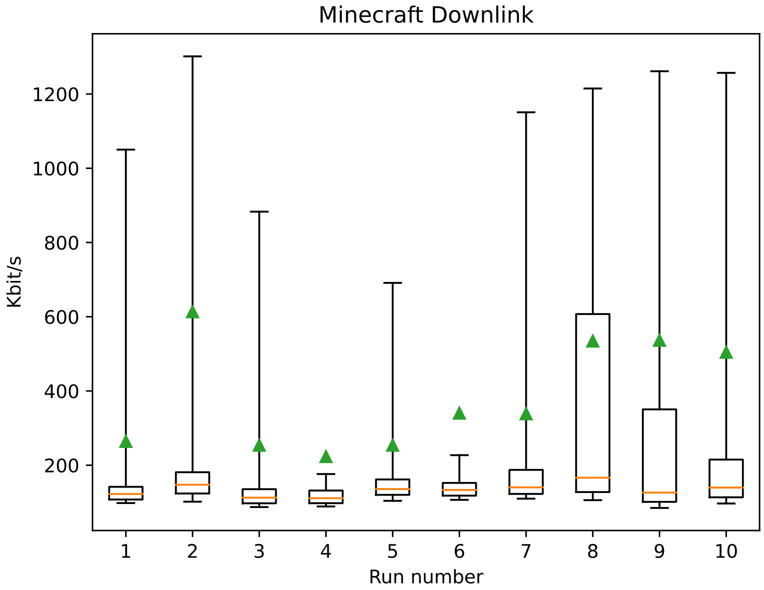


Figure 4.22: Boxplot with Kbit/s on the Downlink with 10 runs of gaming in Minecraft. The boxes are configured with 10th and 90th percentiles

The throughput on the downlink is affected by the extreme deviations in traffic as mentioned in subsection 4.4 covering a single Minecraft game capture. We see that the median value on the downlink is between 100 Kbit/s to 160 Kbit/s. The averages are skewed by the extreme server loads and vary from 200 Kbit/s up to slightly above 600 Kbit/s.

Chapter 5

Discussion

5.1 Scenario 1 - Baseline measurements

5.1.1 Ping Starlink Gateway

The results from scenario 4.3.1 give a solid indication of the RTT over the Starlink network which sits around 40ms most of the time. However, as seen in figure 4.3 it is coherent that the RTT is ramping up and down. A reason for this could be the variation in the satellite distance. Since the satellite is moving at great speeds, the latency probably gets a little higher or lower depending on if the satellite is moving closer, or further away. It is also evident from figure 4.3 that there are some high spikes up to 120ms, this could be because of the satellite hand-offs. Weather data is not measured during this experiment due to clouds and rain not having any impact on the speed of light, and therefore no impact on latency.

5.1.2 Overview over available satellites

From the results in figure 4.5 we see some spikes in the RTT. This happens when the closest satellite is overtaken by another closer satellite which could indicate a hand-off (switching). The study [22] finds packet loss when an overtaking satellite comes closer than the previous closest satellite. A similar tendency could occur when we have spikes in the RTT.

As the satellites travel closer to our position at UiS, we can see a trend in lower RTT, which can indicate a correlation between satellite distance and RTT. The opposite is the case when satellites travel further away from us, which can be seen in figure 4.5 at around 300 seconds.

5.1.3 Constant Bit Rate

We can see from the results in figures 4.6 and 4.7 that the Starlink Satellite broadband service is more than capable of delivering a stable connection with sufficient bandwidth for online gaming. This experiment shows the combination of satellite and terrestrial connection since the server receiving the CBR is located in the UiS.

Included in the plots is the cloud coverage for each run, which tells us that the cloudiness does little to nothing for the packet loss in our case. The median shows us that the packet loss stays mostly at the fixed rate of the selected bandwidth for the experiment, but the average packet loss is above 1% across all runs and bandwidths. From other studies [10], it has been found packet loss is around 0.1% for GEO satellites and LEO (Starlink) around 1.8%. This tells us that our connection with Starlink is on the better side, compared to this study.

5.1.4 Variable Bit Rate

From figure 4.8 it is apparent that BBR outperforms both Reno and Cubic in terms of throughput. The study in [16] about cellular networks informs that it has been tested that BBR outperforms Cubic and NewReno in terms of throughput and latency trade-off. Another paper [22] shows that BBR achieves much higher throughput over Starlink than other CCAs such as Cubic, Reno, Veno, and Vegas. BBR, in this study, achieves a throughput of 55% of the capacity compared to Cubic and Reno which achieve respectively 37% and 30%. The same paper [22] reports that Starlink's satellite hand-offs introduce packet loss. This could explain why there is such a drop in throughput in both Reno and Cubic.

By the nature of how those two (Reno and Cubic) algorithms work, they drop the sending rate of packets based on loss detection. The way Cubic and Reno lower the sending rate is by adjusting the cwnd, when the loss is detected the cwnd is reduced by half. They also have a slow start followed by increasing the sending rate rapidly. Although the two share similar patterns, the way the cwnd is incremented differs. For Reno, this results in a linear recovery state, which means that the cwnd is incremented linearly. For Cubic the recovery state resembles more that of a cubic function, hence the name Cubic. These characteristics for Reno and Cubic can be seen in figure 4.8.

BBR, however, factors in latency rather than packet loss, and attempts to utilize as much bandwidth as possible. If the latency increases significantly it indicates that buffer queues are full and the sending rate of packets needs to be paused until the queues are emptied. This is also referred to as ”draining” the queues. This results in sudden drops in the cwnd which also can be seen in figure 4.8.

5.2 Scenario 2 - Gaming measurements

5.2.1 CS:GO

Based on the results from scenario 4.2.1 it is noticeable that CS:GO does not require much bandwidth. On average for a five-minute run of a classic Deathmatch, the download throughput sits around 300 Kbit/s which translates to 0.3 Mbit/s. From the results in scenario 4.3.3 there is barely any packet loss at 10, 20, and 30 Mbit/s on the downlink. In figure 4.13 we see an average packet loss per ten seconds mostly around one percent. This study [33] says that the limit for acceptable packet loss for FPS games is five percent, which makes Starlink a suitable Internet Service Provider (ISP) for playing CS:GO in terms of packet loss.

The throughput for Starlink compared to terrestrial internet is roughly the same, but there is slightly less on terrestrial as can be seen in figure 4.13. This is probably because the in-game maps that was played were different. The terrestrial test was done only ten km away from UiS, which explains the RRT of around 4ms.

Before investigating further into the RTT, it is worth mentioning that the game server the client connects to is located at University in Stavanger. This means that the traffic travels from UiS through the Starlink satellite network, back to Earth (probably somewhere in Germany based on the angle of the dish), and finally back to UiS through terrestrial internet. The average RTT from UiS to the closest hops near the Starlink gateways in Germany is about 30ms.

In figure 4.9 the RTT results from SPP shows some significant spikes. They seem to appear quite regularly and can be caused by satellite hand-offs as shown in figure 4.5. However, some spikes in SPP data can also be seen in the terrestrial run.

5.2.2 Minecraft

In figure 4.17, Tshark and PsPing data are shown together to illustrate that their trends in RTT values are correlated, thus validating our measurements. Figure 4.16 shows that RTT changes frequently, with highs up to about 125ms and lows at about 30ms. We speculate these changes in RTT may be caused by satellite switching or Starlink gateway switching.

Figure 4.20 shows us the RTT with PsPing and Tshark for all of the Minecraft runs. Here the boxplots display a trend in RTT with values mostly in the region of 65-95ms. With 30ms of deviation between the 10th and the 90th percentile of latency, the distribution of RTT measurements appear promising in regards to Starlink. As seen in a relevant study [19], latency up to 1000ms does not impact gaming QoE, and another study [4] states that a latency of 250ms does start to degrade the performance of the game. Measured against related works, our results prove that Starlink is capable of delivering adequate RTT for Minecraft.

We did compare gaming in Minecraft on both Starlink and terrestrial networks, as can be seen in figure 4.18. This experiment shows us how big of a gap there is in RTT between satellite and terrestrial connection, which is approximately 60-70ms more on the satellite connection. When gaming on the satellite network, we did not feel and noticed the high latency compared to the terrestrial connection. The connection on Starlink did feel as seamless when gaming in Minecraft, as it did on the terrestrial network.

Figure 4.19 shows the throughput Minecraft requires as displayed in the graph of a single experiment run. We see that the average throughput on the downlink stays at around 330 kbit/s which Starlink is more than capable of delivering. This is still the case when the game traffic spikes to almost 2500 kbit/s which occurs when players are exploring and rendering new world data in the game.

Minecraft utilizes an asymmetric downlink/uplink pattern which can be seen in the figure displaying downlink and uplink 4.19 on the same single run. Here we can see that the uplink is far lower than the downlink, which is comparable to the asymmetrical links of the Starlink broadband.

5.2.3 Criteria

Table 5.1 shows a brief overview of the different metrics and their corresponding criteria for CS:GO and Minecraft. It also shows our findings to check if the criteria are satisfied.

Criteria	Related work	Our results	Criteria met?
RTT CS:GO: < 60ms	[33]	83ms	No
RTT Minecraft: < 250ms	[4]	80ms	Yes
Throughput CS:GO: > 400 Kbit/s	4.15	150 Mbit/s	Yes
Throughput Minecraft: > 2400 Kbit/s	4.19	150 Mbit/s	Yes
Packet loss CS:GO: < 5%	[6]	1%	Yes

Table 5.1: QoS metric criteria for CS:GO and Minecraft

Chapter 6

Conclusions

This thesis has evaluated the Starlink broadband service for the following metrics: latency, throughput, and packet loss. These metrics were evaluated in baseline experiments, and through online gaming experiments. This is to establish if Starlink as an ISP meets the QoS criteria for CS:GO and Minecraft, this is shown in table 5.1. Some online gaming experiments over the terrestrial internet have also been done for comparison reasons. Additionally, we did an experiment to figure out how the satellites move in/out of our Starlink dish view. This result showed us the distance of satellites in correlation with a latency experiment in parallel. The testbed used was rewired depending on the different scenarios, and all of the experiments ran over some variation of automated Python scripts.

The results from this thesis offer some answers to the different research questions that were given in the introduction part. They also open up for some more research regarding Starlink as an ISP for online gaming.

6.1 Answering the Research Questions

- RQ1: Does Starlink satisfy latency and bandwidth requirements for online gaming?

Yes. Our results show that for both CS:GO and Minecraft the throughput Starlink provides, is more than enough for playing these games. One paper about Minecraft [4] points out that it tolerates RTTs up to

250ms. Our results show that Minecraft gameplay has on average a RTT of around 80ms over Starlink. A study on CS:GO QoE [33] found that for RRT higher than 60ms, performance is degrading. The results for this thesis show that RTTs for CS:GO over Starlink are averaging around 85ms. However, as mentioned in the discussion section, the game server is located at UiS which adds an additional 30ms. When connecting to official game servers hosted by Valve, CS:GO automatically tries to find a server with the lowest possible latency. This means that the RTT would be substantially lower (probably in the 50-60ms range) when connecting to an official game server, rather than connecting to the private hosted server at UiS.

- RQ2: Does Starlink produce significant packet loss with regard to online gaming?

For CS:GO, yes. In table 5.1 we see that the packet loss criteria for CS:GO is under 5%. Our results have proven that packet loss over Starlink while playing CS:GO is roughly 1%, as shown in figure 4.13.

- RQ3: What are the differences in performance between terrestrial internet access and Starlink?

The clear difference between Starlink and our terrestrial connection is in the latency. Compared to the fiber-based terrestrial connection, Starlink achieves delays that are 60 to 70ms higher. Regarding bandwidth, both connections provide more than enough capacity for online gaming. Lastly, the packet loss on the terrestrial network was close to none while Starlink averaged at about 1%.

6.2 Future directions

To further discover the online gaming performance on Starlink, we propose conducting thorough QoE surveys on online games. Further, we suggest testing the Starlink broadband through a wider selection of online games and game genres.

Appendix A

Instructions to Compile and Run System

Write your Appendix content here.

A.1 Server config files

A.1.1 CS:GO

```
##################################
######## Default Settings ########
##################################
# DO NOT EDIT, ANY CHANGES WILL BE OVERWRITTEN!
# Copy settings from here and use them in either:
# common.cfg - applies settings to every instance.
# [instance].cfg - applies settings to a specific instance.

#### Game Server Settings ####

## Predefined Parameters | https://docs.linuxgsm.com/configuration/
    start-parameters
# https://docs.linuxgsm.com/game-servers/counter-strike-global-
    offensive
gametype="0"
gamemode="0"
gamemodeflags="0"
skirmishid="0"
mapgroup="mg_active"
```

```
ip="0.0.0.0"
port="27015"
clientport="27005"
sourcetvport="27020"
steamport="26901"
defaultmap="de_mirage"
maxplayers="16"
tickrate="64"

## Game Server Login Token (GSLT): Required
# GSLT is required for running a public server.
# More info: https://docs.linuxgsm.com/steamcmd/gslt
gslt=""

## Workshop Parameters | https://developer.valvesoftware.com/wiki/CSGO_Workshop_For_Server_Operators
# To get an API key visit - https://steamcommunity.com/dev/apikey
wsapikey=""
wscollectionid=""
wsstartmap=""

## Server Parameters | https://docs.linuxgsm.com/configuration
/start-parameters#additional-parameters
startparameters="-game csgo -usercon -strictportbind -ip ${ip}
-port ${port} +clientport ${clientport}
+tv_port ${sourcetvport} +sv_setsteamaccount ${gslt}
-tickrate ${tickrate} +map ${defaultmap}
+servercfgfile ${servercfg} -maxplayers_override
${maxplayers} +mapgroup ${mapgroup} +game_type
${gametype} +game_mode ${gamemode} +sv_game_mode_flags
${gamemodeflags} +sv_skirmish_id ${skirmishid}
+host_workshop_collection ${wscollectionid}
+workshop_start_map ${wsstartmap} -authkey ${wsapikey} -nobreakpad"

#### LinuxGSM Settings ####

## LinuxGSM Stats
# Send useful stats to LinuxGSM developers.
# https://docs.linuxgsm.com/configuration/linuxgsm-stats
# (on|off)
stats="off"

## Notification Alerts
```

```
# (on|off)

# Display IP | https://docs.linuxgsm.com/alerts#display-ip
displayip=""

# More info | https://docs.linuxgsm.com/alerts#more-info
postalert="off"

# Discord Alerts | https://docs.linuxgsm.com/alerts/discord
discordalert="off"
discordwebhook="webhook"

# Email Alerts | https://docs.linuxgsm.com/alerts/email
emailalert="off"
email="email@example.com"
emailfrom=""

# Gotify Alerts | https://docs.linuxgsm.com/alerts/gotify
gotifyalert="off"
gotifytoken="token"
gotifywebhook="webhook"

# IFTTT Alerts | https://docs.linuxgsm.com/alerts/ifttt
iftttalert="off"
ifttttoken="accesstoken"
iftttevent="linuxgsm_alert"

# Mailgun Email Alerts | https://docs.linuxgsm.com/alerts/mailgun
mailgunalert="off"
mailgunapiregion="us"
mailguntoken="accesstoken"
mailgundomain="example.com"
mailgunemailfrom="alert@example.com"
mailgunemail="email@myemail.com"

# Pushbullet Alerts | https://docs.linuxgsm.com/alerts/pushbullet
pushbulletalert="off"
pushbullettoken="accesstoken"
channeltag=""

# Pushover Alerts | https://docs.linuxgsm.com/alerts/pushover
pushoveralert="off"
pushovertoken="accesstoken"
```

```
pushoveruserkey="userkey"

# Rocket.Chat Alerts | https://docs.linuxgsm.com/alerts/rocket.chat
rocketchatalert="off"
rocketchatwebhook="webhook"
rocketchattoken=""

# Slack Alerts | https://docs.linuxgsm.com/alerts/slack
slackalert="off"
slackwebhook="webhook"

# Telegram Alerts | https://docs.linuxgsm.com/alerts/telegram
# You can add a custom cURL string eg proxy (useful in Russia) in "curlcustomstring".
# For example "--socks5 ipaddr:port" for socks5 proxy see more in "curl --help".
telegramapi="api.telegram.org"
telegramalert="off"
telegramtoken="accesstoken"
telegramchatid=""
curlcustomstring=""

## Updating | https://docs.linuxgsm.com/commands/update
updateonstart="off"

## Backup | https://docs.linuxgsm.com/commands/backup
maxbackups="4"
maxbackupdays="30"
stoponbackup="on"

## Logging | https://docs.linuxgsm.com/features/logging
consolelogging="on"
logdays="7"

## Monitor | https://docs.linuxgsm.com/commands/monitor
# Query delay time
querydelay="1"

## ANSI Colors | https://docs.linuxgsm.com/features/ansi-colors
ansi="on"

#### Advanced Settings ####

```

```
## Message Display Time | https://docs.linuxgsm.com/features/message-display-time
sleeptime="0.5"

## SteamCMD Settings | https://docs.linuxgsm.com/steamcmd
# Server appid
appid="740"
steamcmdforcewindows="no"
# SteamCMD Branch | https://docs.linuxgsm.com/steamcmd/branch
branch=""
betapassword=""
# Master Server | https://docs.linuxgsm.com/steamcmd/steam-master-server
steammaster="true"

## Stop Mode | https://docs.linuxgsm.com/features/stop-mode
# 1: tmux kill
# 2: CTRL+c
# 3: quit
# 4: quit 120s
# 5: stop
# 6: q
# 7: exit
# 8: 7 Days to Die
# 9: GoldSrc
# 10: Avorion
# 11: end
stopmode="9"

## Query mode
# 1: session only
# 2: gamedig (gsquery fallback)
# 3: gamedig
# 4: gsquery
# 5: tcp
querymode="2"
querytype="protocol-valve"

## Console type
consoleverbose="yes"
consoleinteract="yes"

## Game Server Details
```

```
# Do not edit
gamename="Counter-Strike: Global Offensive"
engine="source"
glibc="2.15"

#### Directories ####
# Edit with care

## Game Server Directories
systemdir="${serverfiles}/csgo"
executabledir="${serverfiles}"
executable="./srcds_run"
servercfgdir="${systemdir}/cfg"
servercfg="${selfname}.cfg"
servercfgdefault="server.cfg"
servercfgfullpath="${servercfgdir}/${servercfg}"

## Backup Directory
backupdir="${lgsmdir}/backup"

## Logging Directories
logdir="${rootdir}/log"
gamelogdir="${systemdir}/logs"
lgsmlogdir="${logdir}/script"
consolelogdir="${logdir}/console"
lgsmlog="${lgsmlogdir}/${selfname}-script.log"
consolelog="${consolelogdir}/${selfname}-console.log"
alertlog="${lgsmlogdir}/${selfname}-alert.log"
postdetailslog="${lgsmlogdir}/${selfname}-postdetails.log"

## Logs Naming
lgsmlogdate="${lgsmlogdir}/${selfname}-script-$(date '+%Y-%m-%d-%H:%M:%
    S').log"
consolelogdate="${consolelogdir}/${selfname}-console-$(date '+%Y-%m-%d
    -%H:%M:%S').log"
```

Listing A.1: default properties for csgo server

```
##################################
####### Instance Settings ########
##################################
```

```
# PLACE INSTANCE SETTINGS HERE
## These settings will apply to a specific instance.
defaultmap="de_dust2"
gslt="7A5801E75278DB8BD7B6D1AFB9BD3883"
gametype="1"
gamemode="2"
gamemodeflags="32"
maxplayers="32"
```

Listing A.2: server properties for csgo server

A.1.2 Minecraft

```
##################################
######## Default Settings ########
##################################
# DO NOT EDIT, ANY CHANGES WILL BE OVERWRITTEN!
# Copy settings from here and use them in either:
# common.cfg - applies settings to every instance.
# [instance].cfg - applies settings to a specific instance.

#### Game Server Settings ####

## Predefined Parameters | https://docs.linuxgsm.com/configuration
/start-parameters
javaram="1024" # -Xmx$1024M

## Server Parameters | https://docs.linuxgsm.com/configuration
/start-parameters#additional-parameters
startparameters="nogui"

## Release Settings | https://docs.linuxgsm.com/game-servers
/minecraft#release-settings
# Branch (release|snapshot)
branch="release"
# Version (latest|1.16)
mcversion="latest"

#### LinuxGSM Settings ####

## LinuxGSM Stats
```

```
# Send useful stats to LinuxGSM developers.
# https://docs.linuxgsm.com/configuration/linuxgsm-stats
# (on|off)
stats="off"

## Notification Alerts
# (on|off)

# Display IP | https://docs.linuxgsm.com/alerts#display-ip
displayip=""

# More info | https://docs.linuxgsm.com/alerts#more-info
postalert="off"

# Discord Alerts | https://docs.linuxgsm.com/alerts/discord
discordalert="off"
discordwebhook="webhook"

# Email Alerts | https://docs.linuxgsm.com/alerts/email
emailalert="off"
email="email@example.com"
emailfrom=""

# Gotify Alerts | https://docs.linuxgsm.com/alerts/gotify
gotifyalert="off"
gotifytoken="token"
gotifywebhook="webhook"

# IFTTT Alerts | https://docs.linuxgsm.com/alerts/ifttt
iftttalert="off"
ifttttoken="accesstoken"
iftttevent="linuxgsm_alert"

# Mailgun Email Alerts | https://docs.linuxgsm.com/alerts/mailgun
mailgunalert="off"
mailgunapiregion="us"
mailguntoken="accesstoken"
mailgundomain="example.com"
mailgunemailfrom="alert@example.com"
mailgunemail="email@myemail.com"

# Pushbullet Alerts | https://docs.linuxgsm.com/alerts/pushbullet
pushbulletalert="off"
```

```
pushbullettoken="accesstoken"
channeltag=""

# Pushover Alerts | https://docs.linuxgsm.com/alerts/pushover
pushoveralert="off"
pushovertoken="accesstoken"
pushoveruserkey="userkey"

# Rocket.Chat Alerts | https://docs.linuxgsm.com/alerts/rocket.chat
rocketchatalert="off"
rocketchatwebhook="webhook"
rocketchattoken=""

# Slack Alerts | https://docs.linuxgsm.com/alerts/slack
slackalert="off"
slackwebhook="webhook"

# Telegram Alerts | https://docs.linuxgsm.com/alerts/telegram
# You can add a custom cURL string eg proxy (useful in Russia) in "curlcustomstring".
# For example "--socks5 ipaddr:port" for socks5 proxy see more in "curl --help".
telegramapi="api.telegram.org"
telegramalert="off"
telegramtoken="accesstoken"
telegramchatid=""
curlcustomstring=""

## Updating | https://docs.linuxgsm.com/commands/update
updateonstart="off"

## Backup | https://docs.linuxgsm.com/commands/backup
maxbackups="4"
maxbackupdays="30"
stoponbackup="on"

## Logging | https://docs.linuxgsm.com/features/logging
consolelogging="on"
logdays="7"

## Monitor | https://docs.linuxgsm.com/commands/monitor
# Query delay time
querydelay="1"
```

```

## ANSI Colors | https://docs.linuxgsm.com/features/ansi-colors
ansi="on"

#### Advanced Settings ####

## Message Display Time | https://docs.linuxgsm.com/features/message-display-time
sleeptime="0.5"

## Stop Mode | https://docs.linuxgsm.com/features/stop-mode
# 1: tmux kill
# 2: CTRL+c
# 3: quit
# 4: quit 120s
# 5: stop
# 6: q
# 7: exit
# 8: 7 Days to Die
# 9: GoldSrc
# 10: Avorion
# 11: end
stopmode="5"

## Query mode
# 1: session only
# 2: gamedig (gsquery fallback)
# 3: gamedig
# 4: gsquery
# 5: tcp
querymode="2"
querytype="minecraft"

## Console type
consoleverbose="yes"
consoleinteract="yes"

## Game Server Details
# Do not edit
gamename="Minecraft"
engine="lwjgl2"
glibc="null"

```

```
#### Directories ####
# Edit with care

## Game Server Directories
systemdir="${serverfiles}"
executabledir="${serverfiles}"
preexecutable="java -Xmx${javaram}M -jar"
executable="./minecraft_server.jar"
servercfgdir="${systemdir}"
servercfg="server.properties"
servercfgdefault="server.properties"
servercfgfullpath="${servercfgdir}/${servercfg}"

## Backup Directory
backupdir="${lgsmdir}/backup"

## Logging Directories
logdir="${rootdir}/log"
gamelogdir="${systemdir}/logs"
lgsmlogdir="${logdir}/script"
consolelogdir="${logdir}/console"
lgsmlog="${lgsmlogdir}/${selfname}-script.log"
consolelog="${consolelogdir}/${selfname}-console.log"
alertlog="${lgsmlogdir}/${selfname}-alert.log"
postdetailslog="${lgsmlogdir}/${selfname}-postdetails.log"

## Logs Naming
lgsmlogdate="${lgsmlogdir}/${selfname}-script-$(date '+%Y-%m-%d-%H:%M:%S').log"
consolelogdate="${consolelogdir}/${selfname}-console-$(date '+%Y-%m-%d-%H:%M:%S').log"
```

Listing A.3: default properties for minecraft server

```
#Minecraft server properties
#Wed Apr 19 09:49:20 CEST 2023
enable-jmx-monitoring=false
level-seed=
rcon.port=27016
enable-command-block=false
gamemode=survival
```

```
enable-query=true
generator-settings={}
enforce-secure-profile=true
level-name=world
motd=LinuxGSM
query.port=27017
pvp=true
generate-structures=true
max-chained-neighbor-updates=1000000
difficulty=easy
network-compression-threshold=256
max-tick-time=60000
require-resource-pack=false
max-players=20
use-native-transport=true
online-mode=true
enable-status=true
allow-flight=false
initial-disabled-packs=
broadcast-rcon-to-ops=true
view-distance=10
max-build-height=256
server-ip=152.94.120.57
resource-pack-prompt=
allow-nether=true
server-port=27015
enable-rcon=false
sync-chunk-writes=true
op-permission-level=4
prevent-proxy-connections=false
hide-online-players=false
resource-pack=
entity-broadcast-range-percentage=100
simulation-distance=10
player-idle-timeout=0
rcon.password=adminvpuplYWH
force-gamemode=false
rate-limit=0
hardcore=false
white-list=false
broadcast-console-to-ops=true
spawn-npcs=true
spawn-animals=true
```

```
snooper-enabled=true
function-permission-level=2
initial-enabled-packs=vanilla
level-type=default
text-filtering-config=
spawn-monsters=true
enforce-whitelist=false
resource-pack-sha1=
spawn-protection=16
max-world-size=29999984
```

Listing A.4: server properties for minecraft server

A.2 Code

A.2.1 Experiments

```python
from setup.game_servers import CSGOServer, MinecraftServer
from setup.iperf_cbr import run_iperf_cbr_udp, filter_iperf_cbr_udp
from setup.yr_api import yr
from setup.config import SERVER_IP, CS_PORT, MINECRAFT_PORT
from setup.loggers import clear_log, log
from setup.capture_traffic import CSGOClientCapture, CSGOServerCapture,
     MinecraftCapture
from setup.game_latency import ping_client_server
from colorama import Fore, Style
from multiprocessing import Process

import time
import sys


if __name__ == "__main__":

    tests = ["iperf_cbr", "iperf_tcp", "csgo_capture", "mc_capture"]
    print("")

    clear_log()

    for i in tests:
        print(Fore.GREEN + i)
```

```python

print(Fore.WHITE + "")

while True:
    test_option = input(Fore.WHITE + "Choose one of the test options above: ").lower().strip()

    if test_option not in tests:
        print(Fore.RED + "Invalid option: " + test_option)
        continue
    log(f"Test option: {test_option}")
    break

print("")

while True:
    test_duration = input(Fore.WHITE + "Enter test duration in seconds: ")

    try:
        test_duration = int(test_duration.strip())
    except ValueError:
        print(Fore.RED + "Invalid duration, not an interger: " + test_duration)
        continue
    log(f"Test duration: {test_duration}")
    break

print("")

while True:
    test_iterations = input(Fore.WHITE + "Enter test iterations: ")

    try:
        test_iterations = int(test_iterations.strip())
    except ValueError:
        print(Fore.RED + "Invalid iteration, not an interger: " + test_iterations)
        continue
    log(f"Test iterations: {test_iterations}")
    break

print("")
```

```

    while True:
        if test_option == "iperf_cbr":

            test_iperf_stream = input("Upload or download test(up/down)?: ").lower()
            if test_iperf_stream == "up":
                revert = True
            elif test_iperf_stream == "down":
                revert = False
            else:
                revert = False

            test_bandwidth = input(Fore.WHITE + "Enter test bandwidth in Mbps: ")
            try:
                test_bandwidth = int(test_bandwidth)
            except ValueError:
                print(Fore.RED + "Invalid bandwidth, not an interger: " + test_iterations)
                continue
            log(f"Test bandwidth: {test_bandwidth}")

        break


    if test_option == "csgo_capture":

        cs_server = CSGOServer("2", "de_dust2")

        log("Starting CSGO-server")

        try:
            cs_server.start_csgo_server()
        except TimeoutError:
            log("FATAL ERROR: Failed to start CSGO-server, check ssh connection")
            sys.exit(0)
        except:
            log("CSGO-server already running")

        input("Press 'Enter' when client is connected to the server: ")

```

```
    log("Client is connected")

    cs_server_capture = CSGOServerCapture(test_duration)
    cs_client_capture = CSGOClientCapture(test_duration)

    time_now_string = f"{time.localtime().tm_mday}-{time.localtime().tm_mon}-{time.localtime().tm_hour}-{time.localtime().tm_min}"

    for i in range(test_iterations):

        if i > 0:
            input("Press 'Enter' when client is connected to the server: ")

        filename = str(time.localtime().tm_mday)+"-"+str(time.localtime().tm_mon)+"-"+str(time.localtime().tm_hour)+"-"+str(time.localtime().tm_min)+"_"+str(i+1)

        try:
            yr(f"csgo_capture_{time_now_string}", i+1)
        except:
            log("FATAL ERROR: Failed to fetch yr weather statistics")

        log(f"Yr api weather measure complete run: {i+1}")

        log(f"Starting CSGO capture on server and client run: {i+1}")

        try:
            cs_server_capture_process = Process(target=cs_server_capture.start_csgo_capture, args=[filename])
            cs_server_capture_process.start()

            cs_client_capture_process = Process(target=cs_client_capture.start_csgo_capture, args=[filename])
            cs_client_capture_process.start()

            ping_process = Process(target=ping_client_server, args=(SERVER_IP, test_duration, filename, CS_PORT))
            ping_process.start()

            cs_server_capture_process.join()
```

```
            cs_client_capture_process.join()
            ping_process.join()

        except TimeoutError:
            log("FATAL ERROR: Failed to capture CSGO-traffic, check ssh connection")
            sys.exit(0)

        log(f"Capture run complete: {i+1}")

    log("FINISHED! CSGO-capture complete")



if test_option == "iperf_cbr":
    for i in range(test_iterations):
        filename = str(time.localtime().tm_mday)+"-"+str(time.localtime().tm_mon)+"-"+str(time.localtime().tm_hour)+"-"+str(time.localtime().tm_min)+"_"+str(i+1)
        run_iperf_cbr_udp(test_bandwidth, test_duration, filename, revert)



if test_option == "mc_capture":

    mc_server = MinecraftServer()

    log("Starting Minecraft-server")

    try:
        mc_server.start_mc_server()
    except TimeoutError:
        log("FATAL ERROR: Failed to start Minecraft-server, check ssh connection")
        sys.exit(0)
    except:
        log("Minecraft-server already running")

    input("Press 'Enter' when client is connected to the server: ")

    mc_capture = MinecraftCapture(test_duration)

```

```
    time_now_string = f"{time.localtime().tm_mday}-{time.localtime().tm_mon}-{time.localtime().tm_hour}-{time.localtime().tm_min}"

    for i in range (test_iterations):

        if i > 0:
            input("Press 'Enter' when client is connected to the server: ")

        try:
            yr(f"minecraft_capture_{time_now_string}", i+1)
        except:
            log("FATAL ERROR: Failed to fetch yr weather statistics")

        log(f"Yr api weather measure complete run: {i+1}")

        log(f"Starting Minecraft run: {i+1}")
        filename = str(time.localtime().tm_mday)+"-"+str(time.localtime().tm_mon)+"-"+str(time.localtime().tm_hour)+"-"+str(time.localtime().tm_min)+"_"+str(i+1)
        mc_filename = "mc_ping_"+filename

        try:
            mc_capture_porcess = Process(target=mc_capture.start_mc_capture, args=[filename])
            mc_capture_porcess.start()

            ping_process = Process(target=ping_client_server, args=(SERVER_IP, test_duration, mc_filename, MINECRAFT_PORT))
            ping_process.start()

            mc_capture_porcess.join()
            ping_process.join()

        except TimeoutError:
            log("FATAL ERROR: Failed to start Minecraft capture, check ssh connection")
            sys.exit(0)



if test_option == "iperf_tcp":
```

```
        pass
```

Listing A.5: Main script

```
import subprocess
import time
import re
import matplotlib.pyplot as _plot
from setup.yr_api import yr

def ping(target, interval, burst_size):

    command = "ping -i %s -c %s %s" % (interval, burst_size, target)
    print(command)

    dict = {}
    errorList = []
    connection = True
    timeOfPing = time.asctime()
    timeOut = 0

    #If connection drops, it retries the ping
    while connection:
        try:
            ping = subprocess.check_output(command, shell=True, stderr=subprocess.STDOUT).decode("utf-8")
            connection = False
        except subprocess.CalledProcessError as e:
            errorList.append(["error",e.stdout,time.asctime()])
            print("error")
            time.sleep(10)
            timeOut += 1
            if timeOut == 4:
                connection = False

    #Split ping data to list
    try:
        splitPingToList = re.split("\s", ping)
    except:
        splitPingToList = []

```

```
    #Add time of ping and errors
    dict["time"] = timeOfPing
    dict["error"] = errorList

    #Add pings to dict
    dict["ping"] = []
    for i in splitPingToList:
        if re.search("^time=", i):
            float_ping = round(float(i[5:]), 1)
            dict["ping"].append(float_ping)
        if re.search("^packet",i):
           dict["loss"] = splitPingToList[splitPingToList.index(i)-1]

    #Get weather information
    try:
        weather = yr()
        dict["cloud"] = str(weather["shortIntervals"][0]["symbol"]["clouds"])
        dict["uv"] = str(weather["shortIntervals"][0]["uvIndex"]["value"])
    except:
        dict["cloud"] = "No data"
        dict["uv"] = "No data"
    return dict


def dayRun(stopdate, stophour):
    nowDate = time.localtime().tm_mday
    nowHour = time.localtime().tm_hour
    counter = 0

    while nowDate < stopdate or nowHour < stophour:
        if time.localtime().tm_hour % 2 == 0:
            if time.localtime().tm_min % 15 == 0:
                #Get dictionary with pings and data
                dict = ping("100.64.0.1", 0.5, 1000)

                file = open(f'{counter}','w')
                file.write(dict["time"] + "\t" + "cloud:" + dict["cloud"] + "\t" + "uv:" + dict["uv"] + "\t" + "loss:" + dict["loss"] + "\n")

                for index in range(len(dict["ping"])):
```

```
                    file.write(str(dict["ping"][index])+ "\n")
                file.close()

                if len(dict["error"]) > 0:
                    file = open(f'error_{dict["time"]}','w')
                    for i in range(len(dict["error"])):
                        file.write(str(dict["error"][i]))
                    file.close()

                counter += 1
                nowDate = time.localtime().tm_mday
                nowHour = time.localtime().tm_hour
            else:
                print("15 min sleep")
                nowDate = time.localtime().tm_mday
                nowHour = time.localtime().tm_hour
                time.sleep(20)
        else:
            print("hour sleep")
            nowDate = time.localtime().tm_mday
            nowHour = time.localtime().tm_hour
            time.sleep(20)
```

Listing A.6: Python script to run ping script

```
import subprocess
import sys

args = sys.argv[1:]

def psping(target, duration, filename):

   pingList = subprocess.run(f"psping -n {duration}s {target} ",stdout=subprocess.PIPE, shell=True).stdout.decode("utf-8")

   file = open(f"psping{filename}.txt","w")

   line = pingList.splitlines()
   for i in line[8:-4]:
      splittedLine = i.split(" ")
      file.write(str(splittedLine[5][:-2]) + "\n")
```

```python

   file.close()

psping(args[0], args[1], args[2])
#psping("152.94.120.57:27015","5","3")
```

Listing A.7: Main script

```python
from multiprocessing import Process
import subprocess
import time
from urllib import request
from setup.config import LATITUDE, LONGTIDUDE, ALTITUDE, DEG, N2YO_API_URL, STARLINK_CATEGORY

def closest_sat():
    key ="xxxxx"
    index = 0
    file = open("satellitesMap.txt", "w")
    while index < 500:
        if time.localtime().tm_sec % 2 == 0:
            index += 1
            response = request.get(f"{N2YO_API_URL}above/{LATITUDE}/{LONGTIDUDE}/{ALTITUDE}/{DEG}/{STARLINK_CATEGORY}&apiKey={key}").json()
            file.write(str(response)+"\n")
    file.close()

def ping():

    file = open("satellitePing.txt", "w")
    runPing =  subprocess.run(f"ping -n 500s 100.64.0.1",stdout=subprocess.PIPE, shell=True).stdout.decode("utf-8")
    file.write(runPing)
    file.close()

def main():

    satellite_map = Process(target=closest_sat)
    satellite_map.start()

```

```
    ping_map = Process(target=ping)
    ping_map.start()

    satellite_map.join()
    ping_map.join()
```

Listing A.8: Python script to run API for satellite distance measurment

A.2.2 Post processing

A.2.2.1 PCAP files

```
import pyshark
from datetime import datetime
import matplotlib.pyplot as plt

pathToFileTcp = "" # Path to tcp pcap file

# Specified path to Pcap-file
pathToFileClient = ""
pathToFileServer = ""
iteration = 10

# Returns a list of packets filtered by provided source and dst IP
    addresses
def get_pkts_by_ip(list, ipSrc, ipDst):
    listPackets = []

    # Assigns source and destination IPs for every packet in a list to
    two variables.
    for packet in list:
        try:
            ipExists = packet.ip.src
        except:
            pass
        try:
            ipExist = packet.ip.dst
        except:
            pass
        # Checks if packet IPs matches IPs we are filtering for
        if ipExists == ipSrc and ipExist == ipDst:
```

```
            listPackets.append(packet)

    return listPackets


# Returns Pcap-file from specified path
def capture(path):
    cap = pyshark.FileCapture(path)
    return cap

# Returns Pcap-file of only TCP retransmission packets
def capture_file_tcp_retransmit(path):
    cap = pyshark.FileCapture(path, display_filter='tcp.analysis.retransmission')
    return cap

# Returns a list of all captured UDP data
def get_udp_data(capFile):
    udpData = []
    for packet in capFile:
        if packet.transport_layer == "UDP":
            udpData.append(packet)
        #might need try/except
    return udpData

# Returns a list of all captured TCP data
def get_tcp_data(capFile):
    tcpData = []
    for packet in capFile:
        if packet.transport_layer == "TCP":
            tcpData.append(packet)
        #might need try/except
    return tcpData

# Writes a single value to the end of a txt file.
# You need to specify your path and fileNum represents the iteration number
def write_single_variable_to_txt(path, variable, fileNum):
    f = open(f"{path}_{fileNum}.txt",  "a+")
    f.write(f"{variable}"+ "\n")
    f.close()
```

```python
# Returns a list of lists where each list contains the amount of packets sent in a single second
def list_of_pkts_per_second(someList):
    megaList = []
    newList = []
    nextSeconds = 0
    currSeconds = 0

    newList.append(someList[0])

    for i in range(len(someList) - 1):
        # Parses the time field of the packet to seconds and compares it to the seconds of the next packet
        currSeconds = int(someList[i].sniff_time.second)
        nextSeconds = int(someList[i+1].sniff_time.second)
        if nextSeconds != currSeconds:
            # Appends the list if they are inequal, empties the concurrent list and appends the next packet to it.
            megaList.append(newList)
            newList = []
            newList.append(someList[i+1])
        else:
            # The packets are in the same second and appends the packet.
            newList.append(someList[i+1])

    return megaList

# Returns a list of lists where each list contains the sum of bytes in a single second
def list_of_bytes_per_second(someList):
    megaList = []
    newList = []
    nextSeconds = 0
    currSeconds = 0

    newList.append(int(someList[0].length))

    for i in range(len(someList) - 1):
        # Parses the time field of the packet to seconds and compares it to the seconds of the next packet
        currSeconds = int(someList[i].sniff_time.second)
        nextSeconds = int(someList[i+1].sniff_time.second)
```

```python
        if nextSeconds != currSeconds:
            # If inequal, sums bytelengths of packets in the concurrent list and appends it, empties the concurrent list and appends the next packet to it.
            sum = 0
            for i in newList:
                sum += i
            megaList.append(sum)
            newList = []
            sum = 0
            newList.append(int(someList[i+1].length))
        else:
            # If equal, appends packet to concurrent list.
            newList.append(int(someList[i+1].length))

    return megaList

# How to process a CSGO run with pcap files from both the client and the server
# It writes bytes per second to txt files of specified paths

if __name__ == "__main__":
    print("Analyzing packets...")

    # IP addresses of the client and of the server. Used to filter out game data
    clientIp = ""
    serverIp = ""

    # Specify paths to write to
    pathServerDataFromServer = ""
    pathServerDataFromClient = ""
    pathClientDataFromServer = ""
    pathClientDataFromClient = ""

    # Read server pcap file and extract UDP data
    csgoDataServer = capture(pathToFileServer)
    csgoUdpDataServer = get_udp_data(csgoDataServer)

    # Read client pcap file and extract UDP data
    csgoDataClient = capture(pathToFileServer)
    csgoUdpDataClient = get_udp_data(csgoDataClient)

```

```
    # Extract packets by specified client and server IPs.
    serverDataFromServer = get_pkts_by_ip(csgoUdpDataServer,serverIp,
clientIp)
    serverDataFromClient = get_pkts_by_ip(csgoUdpDataServer,clientIp,
serverIp)
    clientDataFromServer = get_pkts_by_ip(csgoUdpDataClient,serverIp,
clientIp)
    clientDataFromClient = get_pkts_by_ip(csgoUdpDataClient,clientIp,
serverIp)

    # Get a list of lists of bytes per second for the game data
    serverDataFromServerBytesPerSecond = list_of_bytes_per_second(
serverDataFromServer)
    serverDataFromClientBytesPerSecond = list_of_bytes_per_second(
serverDataFromClient)
    clientDataFromServerBytesPerSecond = list_of_bytes_per_second(
clientDataFromServer)
    clientDataFromClientBytesPerSecond = list_of_bytes_per_second(
clientDataFromClient)


    # Lastly, loop through the list of lists to get the number of bytes
 each second and write it to a txt file.
    # The txt file will be written to the specified path with the run
number in the end of it.
    for listOfBytes in serverDataFromServerBytesPerSecond:
        write_single_variable_to_txt(pathServerDataFromServer,
listOfBytes, iteration)

    for listOfBytes in serverDataFromClientBytesPerSecond:
        write_single_variable_to_txt(pathServerDataFromClient,
listOfBytes, iteration)

    for listOfBytes in clientDataFromServerBytesPerSecond:
        write_single_variable_to_txt(pathClientDataFromServer,
listOfBytes, iteration)

    for listOfBytes in clientDataFromClientBytesPerSecond:
        write_single_variable_to_txt(pathClientDataFromClient,
listOfBytes, iteration)
```

Listing A.9: Processing pcap files

A.2.2.2 Plot

```python
import json
import os
import signal
import statistics
import matplotlib.pyplot as _plot
import numpy as np
import scipy.signal as sc
import pandas as pd
import statistics

#############################
#######Helper functions######
#############################
def return_minecraft_psping_measurment_list(path):
    file = open(path, "r")
    list = []
    for i in file:
        value = i.strip("\n")
        if(float(value) < 250): # filter out outliers
            list.append(float(value))
    return list

def return_minecraft_tshark_measurmentlist(path):
    file = open(path, "r")
    list = []
    lines = file.readlines()
    for i in lines:
        if float(i.strip("\n")) < 130:
            list.append(float(i.strip("\n")))
        else:
            list.append(float(80))
    return list

def psping_terrestrial_minecraft(path):
    file = open(path, "r")
    lines = file.readlines()
    list = []
    for line in lines:
        list.append(float(line.strip("\n")))
    file.close()
```

```

    return list

def tshark_terrestiral_minecraft(path):
    file = open(path, "r")
    lines = file.readlines()
    list = []
    for line in lines:
        list.append(float(line.strip("\n")))
    file.close()

    return list

def return_psping_measurment_list_from_minecraft_runs(path):
    arr = os.listdir(path)
    tcpping_list = []
    for i in arr:
        file = open(f"{path}/{i}", "r")
        list = []
        for i in file:
            value = i.strip("\n")
            if(float(value) < 250):
                list.append(float(value))
        tcpping_list.append(list)
    return tcpping_list


def return_spp_measurment_list_from_minecraft_runs(path):
    arr = os.listdir(path)
    list = []
    for i in arr[7:]:
        file = open(f"{path}/{i}", "r")
        lines = file.readlines()
        list = []
        for line in lines:
            list.append(float(line.strip("\n")))

        list.append(list)
    return list

def return_csgo_psping_measurment_list(path):
    arr = os.listdir(path)
    tcpping_list = []
```

```python
    for i in arr:
        file = open(f"{path}/{i}", "r")
        list = []
        for i in file:
            value = i.strip("\n")
            if(float(value) < 250):
                list.append(float(value))
        tcpping_list.append(list)
    return tcpping_list


def return_csgo_spp_measurment_list(path):
    arr = os.listdir(path)
    tcpping_list = []
    for i in arr:
        file = open(f"{path}/{i}", "r")
        list = []
        for i in file:
            value = i.strip("\n")
            if round(float(value[-9:-1])*1000, 2) < 250:
                list.append(round(float(value[-9:-1])*1000, 2))
        tcpping_list.append(list)
    return tcpping_list

def get_weather_from_cbr_run(path):
    file = open(path, "r")
    line = file.readlines()
    list = []
    counter = 1
    for i in line:
        if counter % 2 == 0:
            a = i.split("\t")
            list.append(int(a[0]))
        counter += 1
    return list

def box_plot(data, edge_color, fill_color):
    bp = _plot.boxplot(data, whis=[10,90],patch_artist=True, showfliers
    =False,showmeans=True)

    for element in ['boxes', 'whiskers', 'fliers', 'means', 'medians',
    'caps']:
        _plot.setp(bp[element], color=edge_color)
```

```python

    for patch in bp['boxes']:
        patch.set(facecolor=fill_color)

    return bp

def CSGO_TCPping_list(path):
    file = open(path, "r")
    tcpping_list = []
    for i in file:
        value = i.strip("\n")
        if(float(value) < 250):
            tcpping_list.append(float(value))

    y = np.array(tcpping_list)
    return y

def get_packet_loss_percent(clientfile, serverfile):
    clientFile_downlink = open(clientfile, "r")
    serverFile_uplink = open(serverfile, "r")

    server_lines = serverFile_uplink.readlines()
    client_lines = clientFile_downlink.readlines()

    client_list = []
    server_list = []

    for i in client_lines:
        i.strip("\n")
        if int(i) > 50:
            client_list.append(int(i))
        else:
            client_list.append(64)

    for i in server_lines:
        i.strip("\n")
        if int(i) > 50:
            server_list.append(int(i))
        else:
            server_list.append(64)

    packet_loss_list = []
    counter = 0
```

```
    num_of_packets_sent = 0
    packet_loss = 0

    while counter < 299:
        for i in range(10): ## Adjust this range to set window length.
            packet_loss += client_list[counter] - server_list[counter]
            num_of_packets_sent += server_list[counter]
            counter += 1
        pl_percent = round(float(packet_loss/num_of_packets_sent*100), 2)
        if pl_percent < 0:
            packet_loss_list.append(0.0)
        else:
            packet_loss_list.append(pl_percent)
        num_of_packets_sent = 0
        packet_loss = 0

    return(packet_loss_list)


#############################
#######Plot functions#######
#############################

####### Baseline ##########################################################################################


def baseline_single_run_to_starlink_gateWay(path): # Create RTT plots for a single run with baseline measurment to starlink gateway
    runNumber = "20"
    file = open(path, "r")
    tcpping_list = []
    for i in file.readlines()[1:]:
        value = i.strip("\n")
        if(float(value) < 250):

            tcpping_list.append(round(float(value),2))

    smoothendGraph = sc.savgol_filter(tcpping_list, window_length=21, polyorder=3, mode="nearest")

    _plot.plot(tcpping_list)
```

```
    _plot.plot(smoothendGraph,"r-",lw=1 )
    _plot.legend(["Raw Ping data", "Smoothed data"])
    _plot.ylabel("RTT (ms)")
    _plot.xlabel("Time (s)")
    _plot.title("RTT to Starlink gateway")
    _plot.xlim(0, 1000)
    _plot.grid()
    _plot.show()

def baseline_all_runs_to_starlink_gateway_box_plot(path): #Create RTT plots for all baseline runs to starlink gateway
    pingsTotalList = []
    dateList = []
    count = 0
    TotalCount = 0

    for i in range(0,32):
        file = open(f"{path}/{i}", "r") # open files
        lines = file.readlines()
        pings = []

        for line in lines[1:]:
            TotalCount += 1
            splittedLine = line.strip("\n")
            if float(splittedLine) < 250:
                pings.append(float(splittedLine))

            if float(splittedLine) > 100:
                count += 1

        pingsTotalList.append(pings)

        for data in lines[0:1]:
            splittedData = data.split("\t")
            date = splittedData[0]
            dateList.append(date[11:19])
        file.close()
    print(f"Count: {count}, TotalCount = {TotalCount}")
    print(dateList)
    _plot.boxplot(pingsTotalList,whis=[10,90],showfliers=False, showmeans=True)
    _plot.xticks(np.arange(1, len(dateList)+1), dateList, rotation=90)
    _plot.xlabel("Time")
```

```
    _plot.ylabel("RTT (ms)")
    _plot.ylim(23,55)
    _plot.title("RTT to Starlink gateway")
    _plot.show()


##### CsGo #########################################################################################
def combine_psping_and_spp_all_csgo_runs():  # Combine RTT measurments from CSGO runs
    pspingList = return_csgo_psping_measurment_list("Ping")
    sppList = return_csgo_spp_measurment_list()

    fig, (ax1, ax2) = _plot.subplots(1, 2)

    ax1.boxplot(pspingList, whis=[10,90], showfliers=False, showmeans=True)
    ax1.set_title("CSGO runs RTT measured with psping")
    ax1.set_ylim(70,100)
    ax1.set_xlabel("Run number")
    ax1.set_ylabel("RTT (ms)")

    ax2.boxplot(sppList, whis=[10,90], showfliers=False, showmeans=True)
    ax2.set_title("CSGO runs RTT measured with SPP")
    ax2.set_ylim(70,100)
    ax2.set_xlabel("Run number")
    ax2.set_ylabel("RTT (ms)")

    _plot.show()

def CSGo_starlink_spp_tcpping_loss_plot(path, path2,path3):

    spp_ping_list = []

    file = open(path, "r")

    lines = file.readlines()
    for line in lines:
        spp_ping_list.append(round(float(line[-9:-1])*1000, 2))

    _plot.rcParams['font.size'] = 13
```

```

    y1 = np.array(spp_ping_list)
    x1 = np.arange(0, 300, 50/2527)
    yhat = signal.savgol_filter(y1, window_length=501, polyorder=3, mode="nearest")

    fig, (ax1, ax2) = _plot.subplots(2, 1)

    y3 = packet_loss_percent(path2,
                             path3)
    x3 = np.arange(5, 305, 10)

    ax3 = ax1.twinx()
    ax3.plot(x3, y3, "o", markersize="4", color="red", )
    ax3.set_ylim(0, 13)
    ax3.set_ylabel("Packet loss(%)")

    ax1.plot(x1, y1, "-", markersize="5", label="SPP")
    ax1.plot(x1, yhat, "-", lw=2, color="lime")
    ax1.plot(x3, y3, "o", markersize="4", color="red", )
    ax1.set_ylabel("RTT (ms)", fontsize=13)
    ax1.grid()
    ax1.set_xlim(0, 300)
    ax1.set_ylim(40, 170)
    ax1.legend(["Raw SPP data", "SPP data smoothed", "Average loss every 10 s"], loc="upper center", fontsize=13)
    ax1.set_xlabel("Time (s)", fontsize=13)
    ax1.set_title("CSGO RTT SPP", fontsize=13)

    y2 = CSGO_TCPping_list("psping27-3-16-11_9.txt")
    x2 = np.arange(0, len(y2))
    yhat2 = signal.savgol_filter(y2, window_length=22, polyorder=3, mode="nearest")

    ax3 = ax2.twinx()
    ax3.plot(x3, y3, "o", markersize="4", color="red")
    ax3.set_ylim(0, 13)
    ax3.set_ylabel("Packet loss(%)")

    ax2.plot(x2, y2, "-", markersize=5)
    ax2.plot(x2, yhat2, "-",lw=2, color="lime")
    ax2.plot(x3, y3, "o", markersize="4", color="red")
    ax2.set_ylabel("RTT (ms)", fontsize=13)
```

```python
    ax2.set_ylim(40,170)
    ax2.set_xlim(0, 300)
    ax2.grid()
    ax2.legend(["Raw TCP ping data", "TCP ping data smoothed", "Average loss every 10 s"], loc="upper center", fontsize=13)
    ax2.set_xlabel("Time (s)", fontsize=13)
    ax2.set_title("CSGO RTT TCP ping", fontsize=13)

    _plot.subplots_adjust(hspace=0.5)
    _plot.show()

def CSGO_terrestrial_spp_tcpping_plot(path):
    tcpping_list = []

    file = open(path, "r")

    lines = file.readlines()
    for line in lines:
        tcpping_list.append(round(float(line[-9:-1])*1000, 2))

    _plot.rcParams['font.size'] = 13

    y1 = np.array(tcpping_list)
    x1 = np.arange(0, 300, 100/6353)
    yhat = signal.savgol_filter(y1, window_length=501, polyorder=3, mode="nearest")

    fig, (ax1, ax2) = _plot.subplots(2, 1)

    ax1.plot(x1, y1, "-", markersize="5", label="SPP")
    ax1.plot(x1, yhat, "-", lw=2, color="lime")
    ax1.set_ylabel("RTT (ms)", fontsize=13)
    ax1.grid()
    ax1.set_xlim(0, 300)
    ax1.set_ylim(0, 20)
    ax1.legend(["Raw SPP data", "SPP data smoothed", "Average loss every 10 s"], loc="upper center", fontsize=13)
    ax1.set_xlabel("Time (s)", fontsize=13)
    ax1.set_title("CSGO RTT SPP", fontsize=13)

    y2 = CSGO_TCPping_list("psping1CSGOClient.txt")
    x2 = np.arange(0, len(y2))
    yhat2 = signal.savgol_filter(y2, window_length=22, polyorder=3,
```

```
    mode="nearest")

    ax2.plot(x2, y2, "-", markersize=5)
    ax2.plot(x2, yhat2, "-",lw=2, color="lime")
    ax2.set_ylabel("RTT (ms)", fontsize=13)
    ax2.set_ylim(0,20)
    ax2.set_xlim(0, 300)
    ax2.grid()
    ax2.legend(["Raw TCP ping data", "TCP ping data smoothed", "Average loss every 10 s"], loc="upper center", fontsize=13)
    ax2.set_xlabel("Time (s)", fontsize=13)
    ax2.set_title("CSGO RTT TCP ping", fontsize=13)

    _plot.subplots_adjust(hspace=0.5)
    _plot.show()

def CSGO_downlink_starlink_terrestrial_rtt_packetloss_plot(client_path, server_path, terr_path, terr_ping_path, sl_ping_path, client_downlink_path, server_uplink_path, terr_downlink_path, terr_uplink_path):
    clientFile = open(client_path, "r")
    serverFile = open(server_path, "r")

    terr_client = open(terr_path, "r")
    terr_ping = open(terr_ping_path, "r")
    sl_ping = open(sl_ping_path,"r")

    lines_client = clientFile.readlines()
    lines_server = serverFile.readlines()
    lines_terr = terr_client.readlines()
    lines_terr_ping = terr_ping.readlines()
    lines_sl_ping = sl_ping.readlines()

    nyList1 = []
    nyList2 = []
    terr_list = []
    terr_ping_list = []
    sl_ping_list = []

    for i in lines_client:
        i.strip("\n")
        if int(i) > 50:
            nyList1.append(float(i)/1000*8)
```

```
        else:
            nyList1.append(64)

    x1 = np.arange(0, len(nyList1), 1)
    y1 = np.array(nyList1)

    for i in lines_server:
        i.strip("\n")
        if int(i) > 50:
            nyList2.append(float(i)/1000*8)
        else:
            nyList2.append(64)

    mean_client = round(statistics.mean(nyList1))

    for i in lines_terr:
        i.strip("\n")
        terr_list.append(float(i)/1000*8)

    for i in lines_terr_ping:
        i.strip("\n")
        terr_ping_list.append(float(i))

    for i in lines_sl_ping:
        i.strip("\n")
        sl_ping_list.append(float(i))

    _plot.rcParams['font.size'] = 13

    fig, ((ax1, ax4),(ax2, ax3),(ax6, ax5)) = _plot.subplots(3, 2,
sharex=True)
    mean_terr = round(statistics.mean(terr_list))

    ax1.plot(x1, y1)

    ax2.set_xlabel("Time(s)", fontsize=13)
    ax1.set_ylabel("Kbit/s", fontsize=13)

    ax1.legend([f"Avg recieved {mean_client} Kbit/s"], fontsize=13, loc
='upper right')
    ax1.set_title("CSGO Starlink downlink", fontsize=13)
    ax1.set_ylim(151, 550)
    ax1.grid()
```

```

    y2 = packet_loss_percent(client_downlink_path,
                             server_uplink_path)
    x2 = np.arange(5, 305, 10)

    ax2.plot(x2, y2, "ro", markersize="3")
    ax2.legend(["Packet loss every 10s"], fontsize=13, loc='upper right')
    ax2.set_ylim(0, 4)
    ax2.set_xlim(0, 300)
    ax2.set_ylabel("Packet loss(%)", fontsize=13)
    ax2.grid()

    y3 = packet_loss_percent(terr_downlink_path,
                             terr_uplink_path)
    x3 = np.arange(5, 305, 10)

    y4 = np.array(terr_list)
    x4 = np.arange(0, 300)

    ax3.plot(x3, y3, "ro", markersize="3")
    ax3.set_ylim(0, 4)
    ax3.set_ylabel("Packet loss(%)", fontsize=13)
    ax3.legend(["Packet loss every 10s"], fontsize=13, loc='upper right')
    ax3.grid()
    ax3.set_xlabel("Time(s)", fontsize=13)

    ax4.set_title("CSGO Terrestrial downlink", fontsize=13)
    ax4.set_ylabel("Kbit/s", fontsize=13)
    ax4.plot(x4, y4)
    ax4.legend([f"Avg recieved {mean_terr} Kbit/s"], fontsize=13, loc='upper right')
    ax4.set_ylim(151,550)
    ax4.set_xlim(0,300)
    ax4.grid()

    y5 = np.array(terr_ping_list)
    x5 = np.arange(0, 300)

    ax5.plot(x5, y5)
    ax5.set_ylabel("RTT(ms)", fontsize=13)
    ax5.set_xlabel("Time(s)", fontsize=13)
```

```
    ax5.legend(["TCP ping 1s interval"])
    ax5.set_ylim(0, 9)
    ax5.grid()

    y6 = np.array(sl_ping_list)
    x6 = np.arange(0, 300)

    ax6.plot(x6, y6)
    ax6.set_ylabel("RTT(ms)", fontsize=13)
    ax6.set_xlabel("Time(s)", fontsize=13)
    ax6.legend(["TCP ping 1s interval"])
    ax6.grid()

    _plot.subplots_adjust(hspace=0)
    _plot.show()

def CSGO_compare_spp_tcpping(sppPath, tcppingPath):

    spp_ping_list = []
    spp_file = open(sppPath, "r")

    lines = spp_file.readlines()
    for line in lines:
        spp_ping_list.append(round(float(line[-9:-1])*1000, 2))

    tcpping_file = open(tcppingPath, "r")

    tcpping_list = []
    for i in tcpping_file:
        value = i.strip("\n")
        if(float(value) < 250):
            tcpping_list.append(float(value))

    y1 = np.array(spp_ping_list)
    x1 = np.arange(0, 300, 50/2527)
    yhat1 = signal.savgol_filter(y1, window_length=1000, polyorder=3, mode="nearest")

    x2 = np.arange(0, len(tcpping_list))
    y2 = np.array(tcpping_list)
    yhat2 = signal.savgol_filter(y2, window_length=22, polyorder=3, mode="nearest")

```

```
    _plot.plot(x1, yhat1, "r-", lw=2)
    _plot.plot(x2, yhat2, "b-", lw=2)
    _plot.legend(["SPP data smoothed", "TCP ping data smoothed"], loc="upper left", fontsize=14)
    _plot.xlabel("Time (s)")
    _plot.ylabel("RTT (ms)")
    _plot.xlim(0,300)
    _plot.ylim(70, 105)

    _plot.grid()
    _plot.show()


##### Minecraft runs ##########################################################################################

def combine_minecraft_single_run_rtt_plots(psPingPath, TsharkPath): # Create RTT plots for a single run with minecraft measurments with different RTT measurment tools
    psPingList = return_minecraft_psping_measurment_list(psPingPath)
    tsharkList = return_minecraft_tshark_measurmentlist(TsharkPath)

    ## Uncomment if wanting terrestrial data
    #tsharkList = RTTTerrestiralMinecraft()
    #tsharkList = pspingTerrestrialMinecraft()

    filteredPsPing = sc.savgol_filter(psPingList, window_length=11, polyorder=3, mode="nearest")

    filteredTshark = sc.savgol_filter(tsharkList, window_length=451, polyorder=3, mode="nearest")

    x2 = np.arange(0,300,300/len(tsharkList))

    medianTCP = statistics.median(tsharkList)
    print(medianTCP)

    # create subplots with the two lists
    fig, (ax1, ax2) = _plot.subplots(2)

    ax1.plot(psPingList)
    ax1.plot(filteredPsPing,"r-",lw=1 )
```

```
    ax1.set_title("Minecraft RTT PsPing")
    ax1.legend(["Psping raw data ","Smoothed data"],loc ="upper right")
    ax1.set_ylim(0,140)
    ax1.set_xlabel("Time (s)")
    ax1.set_ylabel("RTT (ms)")
    ax1.grid(True)
    ax1.set_xlim(0,300)

    ax2.plot(x2,tsharkList)
    ax2.plot(x2,filteredTshark,"r-",lw=1 )

    ax2.set_title("Minecraft RTT TcpPing")
    ax2.set_ylim(0,140)
    ax2.legend(["TCP raw data","Smoothed data"],loc ="upper right")
    ax2.set_xlabel("Time (s)")
    ax2.set_ylabel("RTT (ms)")
    ax2.grid(True)
    ax2.set_xlim(0,300)
    _plot.show()


def combine_spp_and_psping_minecraft_runs(psPingPath, TsharkPath):  # Combine RTT measurments from Minecraft runs
    pspingList = return_psping_measurment_list_from_minecraft_runs(psPingPath)
    sppList = return_spp_measurment_list_from_minecraft_runs(TsharkPath)

    fig, (ax1, ax2) = _plot.subplots(1, 2)

    ax1.boxplot(pspingList, whis=[10,90], showfliers=False, showmeans=True)
    ax1.set_title("Minecraft runs RTT measured with psping")
    ax1.set_ylim(40,100)
    ax1.set_xlabel("Run number")
    ax1.set_ylabel("RTT (ms)")

    ax2.boxplot(sppList, whis=[10,90], showfliers=False, showmeans=True)
    ax2.set_title("Minecraft runs RTT measured from TCP packages")
    ax2.set_ylim(40,100)
    ax2.set_xlabel("Run number")
    ax2.set_ylabel("RTT (ms)")
```

```python

    _plot.show()

def combine_tcpping_and_psping_minecraft_starlink_and_terrestrial():# Combine Terrestrial and Starlink RTT measurments when gaming on Minecraft
    pspingListStarlink = return_minecraft_psping_measurment_list()
    pspingListTerrestrial = psping_terrestrial_minecraft()

    tcpPingStarlink = return_minecraft_tshark_measurmentlist()
    tcpPingTerrestrial = tshark_terrestiral_minecraft()

    smoothendPsPingStarlink = sc.savgol_filter(pspingListStarlink, window_length=11, polyorder=3, mode="nearest")

    smoothendPsPingTerrestrial = sc.savgol_filter(pspingListTerrestrial, window_length=11, polyorder=3, mode="nearest")

    smoothendTcpPingStarlink = sc.savgol_filter(tcpPingStarlink, window_length=451, polyorder=3, mode="nearest")

    smoothendTcpPingTerrestrial = sc.savgol_filter(tcpPingTerrestrial, window_length=451, polyorder=3, mode="nearest")

    x1 = np.arange(0,300,300/len(smoothendTcpPingStarlink))
    x2 = np.arange(0,300,300/len(smoothendTcpPingTerrestrial))

    fig, (ax1, ax2) = _plot.subplots(1, 2)

    ax1.plot(x1,smoothendTcpPingStarlink)
    ax1.plot(x2,smoothendTcpPingTerrestrial,"r-",lw=1 )
    ax1.legend(["Starlink", "Terrestrial"])
    ax1.set_title("RTT TcpPing")
    ax1.set_ylim(0,100)
    ax1.set_xlabel("Time (s)")
    ax1.set_ylabel("RTT (ms)")
    ax1.grid(True)
    ax1.set_xlim(0,300)

    ax2.plot(smoothendPsPingStarlink)
    ax2.plot(smoothendPsPingTerrestrial,"r-",lw=1 )
    ax2.legend(["Starlink", "Terrestrial"])
    ax2.set_title("RTT PsPing")
```

```
    ax2.set_ylim(0,100)
    ax2.set_xlabel("Time (s)")
    ax2.set_ylabel("RTT (ms)")
    ax2.grid(True)
    ax2.set_xlim(0,300)

    _plot.show()


#### Iperf CBR ##########################################################################################

def create_cbr_box_plot_with_clod_coverage(path):  # Create CBR box plot with cloud coverage
    file = open(path, "r")

    line = file.readlines()

    totalList = []
    counter = 0
    intList = []
    intss = 1

    for i in line:
        if counter % 2 == 0:
            intList.append(intss)
            intss += 1
            a = i.strip("\n")
            nylist = json.loads(a)
            totalList.append(nylist)
        counter += 1

    yrList = get_weather_from_cbr_run()

    x = np.arange(1, len(totalList)+1, 1)

    fig, ax1 = _plot.subplots()

    dateList = []
    for i in range(len(totalList)):

        if i > 15:
```

```python
            dateList.append(int(09.00 + i - 24))

        else:
            dateList.append(int(09.00 + i))

    ax2 = ax1.twinx()
    ax1.boxplot(totalList ,whis=[10,90],showfliers=False, showmeans=True)
    ax1.set_title("Constant bitrate with 30 Mbps bitrate")

    ax2.plot(x,yrList,".", color='purple')
    ax2.set_ylabel("Cloud cover")
    ax1.set_xticklabels(dateList, ha="right")

    ax1.set_ylabel("Packet loss % ")
    ax1.set_xlabel("Time at day")
    _plot.show()

##### Uplink & Downlink ##########################################################################################


def byte_rate_all_runs_minecraft(serverUplinkPath, serverDownlinkPath): # Create box plot of byte rate for all runs of Minecraft
    serverUplink = os.listdir(serverUplinkPath)
    serverDownlink = os.listdir(serverDownlinkPath)

    serverUploadTotalList = []
    serverDownloadTotalList = []

    for i in range(len(serverUplink)):

        serverUplinkFile = open(f"{serverUplinkPath}\{serverUplink[i]}", "r")
        serverDownlinkFile = open(f"{serverDownlinkPath}\\{serverDownlink[i]}", "r")

        serverUplinkLines = serverUplinkFile.readlines()
        serverDownlinkLines = serverDownlinkFile.readlines()
        serverUplinkList = []
        serverDownlinkList = []

        for i in serverUplinkLines:
```

```
            i.strip("\n")
            serverUplinkList.append(8*(int(i)/1000))


        for i in serverDownlinkLines:
            i.strip("\n")

            serverDownlinkList.append(8*(int(i)/1024))

        serverUploadTotalList.append(serverUplinkList)
        serverDownloadTotalList.append(serverDownlinkList)

    _plot.boxplot(serverUploadTotalList,whis=[10,90],showfliers=False, showmeans=True)

    _plot.title("Minecraft Downlink")
    _plot.xlabel("Run number")
    _plot.ylabel("Kbit/s")
    _plot.show()


def packet_rate_all_runs_minecraft(serverUplinkPath, serverDownlinkPath):
    serverUplink = os.listdir(serverUplinkPath)
    serverDownlink = os.listdir(serverDownlinkPath)

    serverUploadTotalList = []
    serverDownloadTotalList = []

    for i in range(len(serverUplink)):
        serverUplinkFile = open(f"{serverUplinkPath}\\{serverUplink[i]}", "r")
        serverDownlinkFile = open(f"{serverDownlinkPath}\\{serverDownlink[i]}", "r")

        serverUplinkLines = serverUplinkFile.readlines()
        serverDownlinkLines = serverDownlinkFile.readlines()
        serverUplinkList = []
        serverDownlinkList = []

        for i in serverUplinkLines:
            i.strip("\n")
            serverUplinkList.append(int(i))
```

```python

        for i in serverDownlinkLines:
            i.strip("\n")
            serverDownlinkList.append(int(i))

        serverUploadTotalList.append(serverUplinkList)
        serverDownloadTotalList.append(serverDownlinkList)

    fig, ax = _plot.subplots()

    bp1 = box_plot(serverUploadTotalList, 'black', 'lightgreen')
    bp2 = box_plot(serverDownloadTotalList, 'black', 'lightblue')

    ax.legend([bp1["boxes"][0], bp2["boxes"][0]], ['Downlink', 'Uplink'
    ],loc='upper right')

    ax.set_title('Minecraft Packet rate')
    ax.set_ylim(20,370)
    ax.set_xlabel('Run number')
    ax.set_ylabel('Packets/s')
    _plot.show()
```

Listing A.10: Script for making plots

A.2.2.3 Satellite distance calculation

```python
import ast
import math
from setup.config import LATITUDE, LONGTIDUDE, ALTITUDE

def calculate_distance(satellite_latitude, satellite_longtitude,
    satellite_altitude):

    # convert longitude and latitude to radians
    converted_latitude = math.radians(LATITUDE)
    converted_longitude = math.radians(LONGTIDUDE)
    converted_satellite_latitude = math.radians(satellite_latitude)
    converted_satellite_longtitude = math.radians(satellite_longtitude)

    # Using Haversine formula to calculate the great circle distance
    line_1 = math.sin((converted_satellite_latitude -
```

```
    converted_latitude) / 2) ** 2 + math.cos(converted_latitude) * math
    .cos(converted_satellite_latitude) * math.sin((
    converted_satellite_longtitude - converted_longitude) / 2) ** 2

    line_2 = 2 * math.atan2(math.sqrt(line_1), math.sqrt(1 - line_1))

    radius_earth = 6371  # Radius of earth in km
    distance_earth = radius_earth * line_2

    # Calculate distance from earth to satellite
    radius_earth = 6371 + ALTITUDE  # Radius of the Earth plus altitude
     of first point
    radius_satellite = radius_earth + satellite_altitude  # Radius of
    the satellite plus altitude of second point
    distance_space = math.sqrt(radius_earth ** 2 + radius_satellite **
    2 - 2 * radius_earth * radius_satellite * math.cos(distance_earth))

    return distance_space

def get_sat(path): # get overview over all satellites in the file that
    is closest

    file = open(path, "r")

    lines = file.readlines()


    distanseList = []
    nameList = []
    listOverSatelliteNames = []

    for index in range(len(lines)):
        findAboveIndex = lines[index].find("above")

        satelliteList = lines[index][findAboveIndex+8:-2]
        res = ast.literal_eval(satelliteList)

        if float(res[i]["satlat"]) < LATITUDE:
            distanse = calculate_distance(res[i]["satlat"], res[i]["
    satlng"], res[i]["satalt"])
            distanseList.append(distanse)
            nameList.append(res[i]["satname"])

```

```
            if distanse < lowestDistanse:
                lowestDistanse = distanse
                lowestSatname = res[i]["satname"]
                listOverSatelliteNames.append(f"{res[i]['satname']} +{index} ")
            distanseList.sort()

        distanseRef = 1000000
        satname = ""

        for i in range(len(distanseList)):
            if distanseList[i] < distanseRef:
                distanseRef = distanseList[i]
                satname = nameList[i]

        listOverSatelliteNames.append(f"{satname} is {distanseRef} km away at time: + {index}  ")

        fila = open("satellitesCalculated.txt", "w")

        for i in range(len(listOverSatelliteNames)):
            print(listOverSatelliteNames[i])
            fila.write(listOverSatelliteNames[i] + "\n")

            fila.close()
```

Listing A.11: Script for calculating satellite distance

A.2.3 Setup

```
from fabric import Connection

class CSGOServerCapture:

    def __init__(self, duration):
        self.duration = duration
        self.test_option = "csgo_server_capture"


    def server_connection(self,user,host,port,key_path):
        with Connection(host=host, user=user, port=port, connect_kwargs
```

```
    ={'key_filename': {key_path}}) as connection:
            return connection


    def start_csgo_capture(self, filename):
        with self.server_connection() as sl_server:
            sl_server.run(f'tshark -i eno1 -w /home/ss/server/TCPDUMP/
    CSGOoutput/{self.test_option}_{filename}.pcap -f "udp or tcp
    portrange 27005-27015" -a duration:{self.duration}')

class CSGOClientCapture:

    def __init__(self, duration):
        self.duration = duration
        self.test_option = "csgo_client_capture"


    def client_connection(self,host,user,port,key_path):
        with Connection(host=host, user=user, port=port, connect_kwargs
    ={'key_filename': {key_path}}) as connection:
            return connection


    def start_csgo_capture(self, filename):
        with self.client_connection() as sl_client:
            sl_client.run(f'tshark -i 4 -w C:/Users/masth/Desktop/{self
    .test_option}_{filename}.pcap -f "udp or tcp portrange 27005-27015"
     -a duration:{self.duration}')

class MinecraftCapture:

    def __init__(self, duration):
        self.duration = duration
        self.test_option = "mc_capture"


    def server_connection(self,host,user,port,key_path):
        with Connection(host=host, user=user, port=port, connect_kwargs
    ={'key_filename': {key_path}}) as connection:
            return connection


    def start_mc_capture(self, filename):
```

```
        with self.server_connection() as sl_server:
            sl_server.run(f'tshark -i eno1 -w /home/ss/server/TCPDUMP/output/{self.test_option}_{filename}.pcap -f "tcp port 27015" -a duration:{self.duration}')
```

Listing A.12: Python script to capture network traffic

```
IPERF3_PORT = 27050
IPERF_HOST_IP = '152.94.120.57'
IPERF_CBR_PATH = 'ENTER PATH TO IPERF CBR LOGS HERE'

YR_PATH = 'ENTER PATH FOR YR CAPTURE FILE HERE'
YR_API_URL = "https://yr.no/api/v0/locations/10-991227/forecast?"

LOG_PATH = 'ENTER PATH TO LOGS HERE'

SERVER_IP = '152.94.120.57'
CLIENT_IP = '192.168.1.154'

CS_PORT = '27015'
MINECRAFT_PORT = '27015'

LONGTIDUDE = 5.733107
LATITUDE = 58.969975
ALTITUDE = 63.0
DEG = 65
N2YO_API_URL = 'https://api.n2yo.com/rest/v1/satellite/'
STARLINK_CATEGORY = 52
```

Listing A.13: Python config

```
from fabric import Connection
from config import CLIENT_IP

def ping_client_server(target,user, duration, filename, port):
    with Connection(host=CLIENT_IP, user=user, port=port, connect_kwargs={'key_filename': 'ENTER KEY PATH'}) as connection:
        connection.run(f"python3 psping.py {target}:{port} {duration} {
```

```
    filename}")
```

Listing A.14: Python script to ssh to client

```python
from fabric import Connection

class CSGOServer:

    def __init__(self, game_mode, map, tick_rate=64):
        self.game_mode = game_mode
        self.map = map
        self.tick_rate = tick_rate

    def server_connection(self,host,user,port,key_path):
        with Connection(host=host, user=user, port=port, connect_kwargs={'key_filename': {key_path}}) as connection:
            return connection


    def start_csgo_server(self):
        with self.server_connection() as sl_server:
            with sl_server.cd("server"):
                sl_server.run(f"bash csgostart.bash {self.map}")


    def stop_csgo_server(self):
         pass

class MinecraftServer:

    def __init__(self) -> None:
        pass

    def server_connection(self,host,user,port,key_path):
        with Connection(host=host, user=user, port=port, connect_kwargs={'key_filename': {key_path}}) as connection:
            return connection


    def start_mc_server(self):
        with self.server_connection() as sl_server:
```

```python
            with sl_server.cd("mcjaca"):
                sl_server.run("./mcserver start")
```

Listing A.15: Python script to connect and start game server

```python
from config import IPERF3_PORT, IPERF_HOST_IP, IPERF_CBR_PATH
from loggers import log
import os


def run_iperf_cbr_udp(bandwidth, duration, filename, revert=False):

    if revert:
        log(f"Running Iperf CBR command download: iperf3 -c {IPERF_HOST_IP} -p {IPERF3_PORT} -b {bandwidth} -t {duration} -u -R --logfile {IPERF_CBR_PATH}{filename}")
        iperf_run = os.system(f"iperf3 -c {IPERF_HOST_IP} -p {IPERF3_PORT} -b {bandwidth} -t {duration} -u -R --logfile {IPERF_CBR_PATH}{filename}")
    else:
        log(f"Running Iperf CBR command upload: iperf3 -c {IPERF_HOST_IP} -p {IPERF3_PORT} -b {bandwidth} -t {duration} -u --logfile {IPERF_CBR_PATH}{filename}")
        iperf_run = os.system(f"iperf3 -c {IPERF_HOST_IP} -p {IPERF3_PORT} -b {bandwidth} -t {duration} -u --logfile {IPERF_CBR_PATH}{filename}")
```

Listing A.16: Python script to do iPerf

```python
from os.path import exists
from config import LOG_PATH

import os

def log(log_string):
    if log_string == "":
        return

    log_file = open(LOG_PATH+"logfile.log", "a+")
```

```python
    log_file.write(log_string+"\n")
    log_file.close()


def clear_log():
    if exists(LOG_PATH+"logfile.log"):
        os.remove(LOG_PATH+"logfile.log")
```

Listing A.17: Python script to start loggers

```python
from config import YR_PATH, YR_API_URL
import requests

def yr(filename, iteration):

    response = requests.get(YR_API_URL).json()

    cloud_value = response["shortIntervals"][0]["symbol"]["clouds"]
    uv_value = response["shortIntervals"][0]["uvIndex"]["value"]
    precip = response["shortIntervals"][0]["symbol"]["precip"]

    file = open(YR_PATH+filename, "a+")

    file.write(str(cloud_value)+"\t")
    file.write(str(uv_value)+"\t")
    file.write(str(precip)+"\t")
    file.write(str(iteration)+"\n")
    file.close()

    return
```

Listing A.18: Python script for weather API request

Bibliography

[1] Activision. *CALL of DUTY*. URL: `https://www.callofduty.com/`. (accessed: 13.04.2023).

[2] Centre for Advanced Internet Architecture. *Synthetic Packet Pairs (SPP) - Tool for passive round trip time measurement*. URL: `http://caia.swin.edu.au/tools/spp/`. (accessed: 30.04.2023).

[3] The European Space Agency. *Types of orbits*. URL: `https://www.esa.int/Enabling_Support/Space_Transportation/Types_of_orbits?fbclid=IwAR1-awivuJGcjV0cPnldsU-WrabbX4t2Zw145iT0E15C97_lijI-Q_gcP0g#GE0`. (accessed: 17.04.2023).

[4] Trevor Alstad et al. "Minecraft computer game simulation and network performance analysis." In: Nov. 2014.

[5] G. Klaus B. Wolfgang. "Go for Gigabit? First Evidence on Economic Benefits of (Ultra-)Fast Broadband Technologies in Europe". In: *Discussion Paper No. 18-020* (Apr. 2018), pp. 0-37. ISSN: 18-020. URL: `http://ftp.zew.de/pub/zew-docs/dp/dp18020.pdf`.

[6] Tom Beigbeder et al. "The Effects of Loss and Latency on User Performance in Unreal Tournament 2003®". In: *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*. NetGames '04. Portland, Oregon, USA: Association for Computing Machinery, 2004, pp. 144–151. ISBN: 158113942X. DOI: `10.1145/1016540.1016556`. URL: `https://doi.org/10.1145/1016540.1016556`.

[7] The SciPy community. *scipy*. URL: `https://docs.scipy.org/doc/scipy/index.html`. (accessed: 20.04.2023).

[8] Valve Corporation. *Counter-Strike: Global Offensive*. URL: `https://store.steampowered.com/app/730/CounterStrike_Global_Offensive/`. (accessed: 13.04.2023).

[9] BENNETT CYPHERS and ERNESTO FALCON. *Fiber*. URL: `https://www.eff.org/deeplinks/2019/10/why-fiber-vastly-superior-cable-and-5g`. (accessed: 07.05.2023).

[10] Joerg Deutschmann, Kai-Steffen Hielscher, and Reinhard German. "Broadband Internet Access via Satellite: Performance Measurements with different Operators and Applications". In: *Broadband Coverage in Germany; 16th ITG-Symposium*. 2022, pp. 1–7.

[11] Joerg Deutschmann et al. "Broadband Internet Access via Satellite: State-of-the-Art and Future Directions". In: *Broadband Coverage in Germany; 15th ITG-Symposium*. 2021, pp. 1–7.

[12] Blizzard Entertainment. *World of Warcraft*. URL: `https://worldofwarcraft.blizzard.com/en-us/`. (accessed: 13.04.2023).

[13] Jeff Forcier. *Fabric*. URL: `https://www.fabfile.org/`. (accessed: 20.04.2023).

[14] Riot Games. *VALORANT*. URL: `https://playvalorant.com/en-gb/`. (accessed: 13.04.2023).

[15] Rockstar Games. *Grand Theft Auto Online*. URL: `https://www.rockstargames.com/gta-online`. (accessed: 13.04.2023).

[16] Carlo Augusto Grazia et al. "BBR+: improving TCP BBR Performance over WLAN". In: *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. 2020, pp. 1–6. DOI: `10.1109/ICC40277.2020.9149220`.

[17] Jim Griner et al. *Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations*. RFC 3135. June 2001. DOI: `10.17487/RFC3135`. URL: `https://www.rfc-editor.org/info/rfc3135`.

[18] Aditi Gupta and Subrat Kar. "Analysis of Packet Aggregation on Cloud Games". In: 2020. DOI: `10.1109/INDICON49873.2020.9342583`.

[19] Oliver Hohlfeld et al. "Insensitivity to Network Delay: Minecraft Gaming Experience of Casual Gamers". In: *2016 28th International Teletraffic Congress (ITC 28)*. Vol. 03. 2016, pp. 31–33. DOI: `10.1109/ITC-28.2016.313`.

[20] Iperf3. *Iperf3 user docs*. URL: `https://iperf.fr/iperf-doc.php`. (accessed: 16.04.2023).

[21] Iperf3. *What is iPerf/iPerf3?* URL: `https://iperf.fr/`. (accessed: 16.04.2023).

[22] Mohamed M. Kassem et al. "A Browser-Side View of Starlink Connectivity". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. IMC '22. Nice, France: Association for Computing Machinery, 2022, pp. 151–158. ISBN: 9781450392594. DOI: `10.1145/3517745.3561457`. URL: `https://doi.org/10.1145/3517745.3561457`.

[23] KimiNewt. *pyshark*. URL: `https://github.com/KimiNewt/pyshark`. (accessed: 20.04.2023).

[24] LinuxGSM. *LinuxGSM*. URL: `https://linuxgsm.com//`. (accessed: 15.04.2023).

[25] Jagex Ltd. *WELCOME TO THE WORLD OF RUNESCAPE*. URL: `https://play.runescape.com/`. (accessed: 13.04.2023).

[26] François Michel et al. "A First Look at Starlink Performance". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. IMC '22. Nice, France: Association for Computing Machinery, 2022, pp. 130–136. ISBN: 9781450392594. DOI: `10.1145/3517745.3561416`. URL: `https://doi.org/10.1145/3517745.3561416`.

[27] Mojang. *WELCOME TO THE OFFICIAL SITE OF MINECRAFT*. URL: `https://www.minecraft.net/en-us`. (accessed: 13.04.2023).

[28] N2YO. *N2YO.COM REST API v1*. URL: `https://www.n2yo.com/api/`. (accessed: 07.05.2023).

[29] R. Nakatsu et al. "QoE and Latency Issues in Networked Games". In: *Handbook of Digital Games and Entertainment Technologies* (2015), pp. 1–36. DOI: `10.1007/978-981-4560-52-8_23-1`.

[30] Nintendo. *Super Smash Bros. Ultimate*. URL: `https://www.smashbros.com/en_GB/index.html`. (accessed: 13.04.2023).

[31] MET Norway. *MET Norway*. URL: `https://api.met.no/weatherapi/`. (accessed: 25.04.2023).

[32] NumFOCUS. *numpy*. URL: `https://numpy.org/`. (accessed: 20.04.2023).

[33] Peter Quax et al. "Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game". In: NetGames '04. Portland, Oregon, USA: Association for Computing Machinery, 2004, pp. 152–156. ISBN: 158113942X. DOI: `10.1145/1016540.1016557`. URL: `https://doi.org/10.1145/1016540.1016557`.

[34] Michael Ray. *online gaming*. URL: `https://www.britannica.com/technology/online-gaming`. (accessed: 10.05.2023).

[35] Michal Ries, Philipp Svoboda, and Markus Rupp. "Empirical study of subjective quality for Massive Multiplayer Games". In: *2008 15th International Conference on Systems, Signals and Image Processing*. 2008, pp. 181–184. DOI: `10.1109/IWSSIP.2008.4604397`.

[36] Mark Russinovich. *PsPing v2.12*. URL: `https://learn.microsoft.com/en-us/sysinternals/downloads/psping`. (accessed: 17.04.2023).

[37] Scapy. *Welcome to Scapy*. URL: `https://scapy.net/`. (accessed: 20.04.2023).

[38] Geographic Scope and Forecast. *Gaming market*. URL: `https://www.verifiedmarketresearch.com/product/gaming-market/`. (accessed: 15.04.2023).

[39] Servers.com. *Differences between peer to peer and dedicated game server hosting*. URL: `https://www.servers.com/news/blog/differences-between-peer-to-peer-and-dedicated-game-server-hosting`. (accessed: 13.04.2023).

[40] Starlink. *Live Starlink Satellite and Coverage Map*. URL: `https://satellitemap.space/`. (accessed: 13.04.2023).

[41] Starlink. *Order Starlink*. URL: `https://www.starlink.com`. (accessed: 7.05.2023).

[42] Starlink. *Starlink*. URL: `https://satellitemap.space/?constellation=starlink`. (accessed: 25.04.2023).

[43] Starlink. *WORLD'S MOST ADVANCED BROADBAND SATELLITE INTERNET*. URL: `https://www.starlink.com/technology`. (accessed: 25.04.2023).

[44] Steam. *GLST*. URL: `https://steamcommunity.com/dev/managegameservers`. (accessed: 15.04.2023).

[45] The Matplotlib development team. *Matplotlib*. URL: `https://matplotlib.org/`. (accessed: 20.04.2023).

[46] tynet.eu. *Sarlink Statuspage*. URL: `https://www.starlinkstatus.space`. (accessed: 10.05.2023).

[47] Valve. *SteamCMD*. URL: `https://developer.valvesoftware.com/wiki/SteamCMD`. (accessed: 15.04.2023).

[48] Chris Veness. *Calculate distance, bearing and more between Latitude/Longitude points*. URL: `https://www.movable-type.co.uk/scripts/latlong.html`. (accessed: 10.05.2023).

[49] Windows. *Minecraft Server*. URL: `https://minecraft.net/en/download/server`. (accessed: 15.04.2023).

[50] Wireshark. *Tshark Manual Page*. URL: `https://www.wireshark.org/docs/man-pages/tshark.html`. (accessed: 20.04.2023).

[51] Xiaokun Xu, Shengmei Liu, and Mark Claypool. “The Effects of Network Latency on Counter-strike: Global Offensive Players”. In: *2022 14th International Conference on Quality of Multimedia Experience (QoMEX)*. 2022, pp. 1–6. DOI: `10.1109/QoMEX55416.2022.9900915`.

[52] 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`.

University
of Stavanger

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no