Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

INTERDISCIPLINARY PROJECT IN ELECTRICAL ENGINEERING

C++-based MASQUE-Proxying for Lower OSI-Layer Protocol Traffic

Christoph Rotte

Technical University of Munich

Department of Informatics

Interdisciplinary Project in Electrical Engineering

C++-based MASQUE-Proxying for Lower OSI-Layer Protocol Traffic

C++-basiertes MASQUE-Proxying für Protokollverkehr der unteren OSI-Schichten

Author:	Christoph Rotte
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Lion Steger, Richard von Seck
Date:	January 15, 2024

I confirm that this Interdisciplinary Project is my own work and I have documented all sources and material used.

Garching, January 15, 2024

Location, Date

Signature

Abstract

Proxy technologies are essential for addressing critical challenges like circumvention of geo-blocking restrictions, resisting online censorship, and enhancing user privacy on the Internet. However, traditional proxying methods have demonstrated limitations, including the issue of head-of-line (HOL) blocking that can constrain performance. Especially this problem in the context of TCP-based proxying has motivated the development of more advanced proxy protocols such as the Multiplexed Application Substrate over QUIC Encryption (MASQUE) protocol, which aims to leverage the capabilities of QUIC and HTTP/3 for superior proxying capabilities. While MASQUE offers theoretical advantages, implementation experience remains limited, presenting an opportunity for empirical analysis of its real-world performance and efficiency.

This project focuses on the development and evaluation of a new MASQUE client and server implementation using Facebook's mvfst and proxygen libraries, which are popular options for QUIC and HTTP/3 functionality. The experiments incorporate multiple nested tunnels. Performance metrics gathered included throughput, time-to-first byte (TTFB), round-trip time (RTT), latency, and jitter. Tests also assessed page load times and CPU usage under different scenarios.

The results from the experiments indicated that the throughput experiences a decrease with each additional proxy hop, which is expected due to the increased encapsulation overhead and the number of processing stages involved. However, it is important to note that while the decrease in throughput is a natural consequence of additional hops, the focus of the findings is on the scalability and efficiency of the proxying process. The implementation demonstrated that it could handle multiple nested tunnels without excessive CPU constraints, showcasing the potential of MASQUE for efficient proxying.

Despite the increase in TTFB, RTT, latency, and jitter with more proxy hops, the TTFB remained relatively stable with a higher number of transactions, suggesting that the MASQUE implementation can handle increased loads effectively. Page load times were sensitive to the number of hops and traffic levels, which is a valuable insight for optimizing MASQUE implementations in scenarios where user experience is critical.

Overall, this analysis provides real-world insights into the capabilities of the MASQUE protocol based on experiments using mvfst and proxygen libraries. It confirms the definite impact of multiple proxy hops [die erkenntnis sollte nicht sein, dass ein proxy die connection langsamer macht, sondern dass die anzahl der hops entscheidend ist. auSSerdem sollte die erkenntnis sein, dass das proxying überhaupt skalierend und effizient ist] on reducing performance for metrics like throughput and latency. The findings con-

tribute to knowledge in the domain of proxying protocols and help motivate refinements to optimize MASQUE implementations going forward.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the current digital era, there is an escalating demand for proxying technologies. Beyond simply enhancing network performance and increasing security, proxies have become essential in addressing diverse global challenges. For instance, they play a role in circumventing geo-blocking restrictions, where access to content is restricted based on location, and in resisting government-imposed online censorship. Furthermore, proxies provide an added layer of online privacy for users in an age where digital surveillance is on the rise [1]. However, the traditional TCP-based proxying methods are showing their limitations. This has encouraged the development of advanced alternatives. Among these, the Multiplexed Application Substrate over QUIC Encryption (MASQUE) protocol is notable as it harnesses the innovations of QUIC and HTTP/3 to provide superior proxying capabilities [2], [3].

1.1 Motivation

Despite the advancements in proxying protocols and the development of QUIC, there is still a need for a unified solution that can efficiently proxy various types of data over a single connection while avoiding performance problems like Head-of-Line (HOL) blocking. This is where MASQUE [4] comes into play. MASQUE is designed to adapt the general approach of HTTP CONNECT [5] for HTTP/3, leveraging the benefits of QUIC to provide a more efficient and versatile proxying solution for both UDP and IP traffic.

Although HTTP/3 is the main target for MASQUE, fallbacks for both HTTP/1 and HTTP/2 are provided to ensure compatibility and maintain performance improvements

across different versions of HTTP. This approach ensures that MASQUE can be utilized in various scenarios, even when HTTP/3 is not available or fully supported.

The MASQUE protocol is currently being specified by the MASQUE Working Group, an IETF task force, with the aim of developing and standardizing the protocol to ensure compatibility, efficiency, and security. [6]

MASQUE is still in its early stages with both main protocols only recently standardized: CONNECT-UDP since August 2022 (RFC 9298) [2] and CONNECT-IP (RFC 9484) [7] since October 2023. The implementation status of MASQUE is also in the early stages, with Google's QUICHE[1] library being one of the first large libraries to start implementing MASQUE proxying.

Note that the work for this IDP took place parallel to the developement of RFC 9484, starting with [8], which is why some details may lean on older drafts of CONNECT-IP.

1.2 Research Questions

The primary goal of our implementation of the MASQUE protocol is to evaluate three distinct areas.
Firstly, we seek to understand the implications of encapsulation overhead, i.e. the impact of using multiple header layers, on the efficiency and operation of the protocol.
Secondly, we will assess the transmission performance, considering data transfer rates and the overall reliability of communications.
Lastly, we'll address library-specific differences and challenges, investigating how different libraries might affect the MASQUE protocol's implementation and functionality.

Existing MASQUE implementations have not yet captured all the elements we need for this thorough evaluation. This has led us to develop a custom version tailored to address these specific research questions more effectively.

[1] `https://github.com/google/quiche`

Chapter 2

Background

Conventional proxying techniques, including SOCKet Secure (SOCKS) [9] and HTTP CONNECT, mainly run on top of TCP, a reliable and extensively utilized transport protocol. Although TCP offers a robust basis for proxying, it introduces constraints that affect the performance and effectiveness of proxy applications. One such constraint is the HOL blocking issue, where the transfer of data across multiple streams can be limited due to the ordered delivery of packets [10]. As the necessity for high-performance proxying keeps expanding, there is a need for solutions that can tackle these limitations and pave the way for enhanced proxying technologies [11].

2.1 TCP Proxying

Managing connections between clients and servers in network applications has traditionally relied on TCP-based proxying. TCP offers reliable, connection-oriented communication that ensures data is transmitted correctly and in the right order. However, this approach also has limitations, particularly in the context of proxying. When multiple logical streams are reduced to a single connection in a TCP-based proxy setup, it can result in HOL blocking. This, in turn, can lead to slowdowns across multiple streams, affecting the efficiency and performance of the proxy application.

To illustrate this point, consider the visual representation of HOL blocking in TCP-based proxying shown in Figure 2.1. This diagram depicts three logically separated streams running concurrently over an HTTP/2 CONNECT connection. HTTP/2 CONNECT is a method used to establish network connections through an HTTP/2 proxy, allowing clients to form a secure and reliable TCP-based tunnel through the proxy for communicating with remote servers.

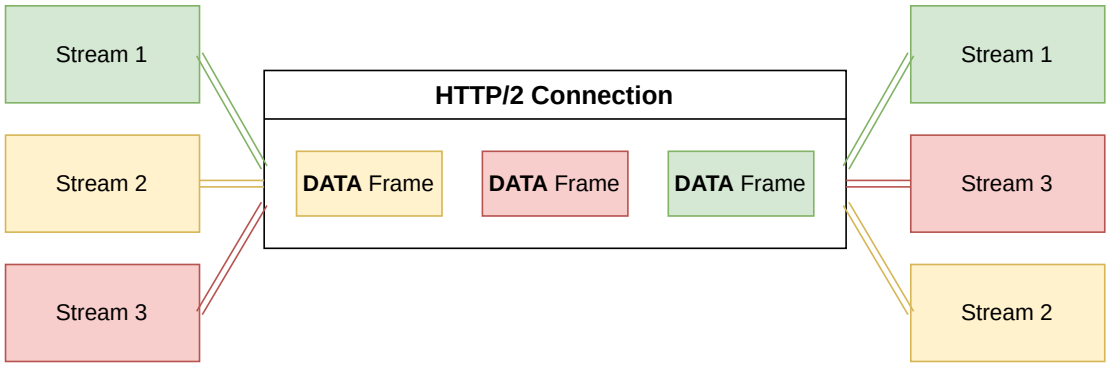

Figure 2.1: TCP Proxying via HTTP/2 CONNECT

In the figure, let us focus on two particular packets: the red packet and the yellow packet. The red packet, due to some issues such as network congestion or packet loss, gets stuck and cannot be delivered promptly. Since TCP insists on delivering packets in order, the subsequent yellow packet, even though it is ready, cannot be transmitted until the red packet has been successfully delivered. This means that the yellow packet, and potentially others following it, will experience delays because of the hold-up caused by the red packet. This phenomenon where one stream's packets can block another's is a clear demonstration of HOL blocking in TCP-based proxying.

The figure highlights the negative impact of HOL blocking on the performance of TCP-based proxying. This effect can cause slowdowns across multiple streams, regardless of their individual transmission rates or the availability of their packets. In order to be able to scale with the number of streams, it would be necessary to create multiple TCP connections, which is more expensive than running multiple streams over the same connection.

The performance issues associated with HOL blocking increase with the number of streams. As a result, alternative transport protocols and proxying methods, such as QUIC and the MASQUE protocol, have been developed to address these challenges and provide a better foundation for proxying technologies.

2.2 QUIC

QUIC [12] is a new transport layer protocol developed to fix issues seen with TCP, with the aim to make web applications run faster and smoother. Built upon UDP, QUIC offers advantages like reduced connection establishment time, enhanced congestion control, and robust security provisions. The protocol forms the foundation for HTTP/3 [13], marking a significant departure from the TCP-based HTTP/2.

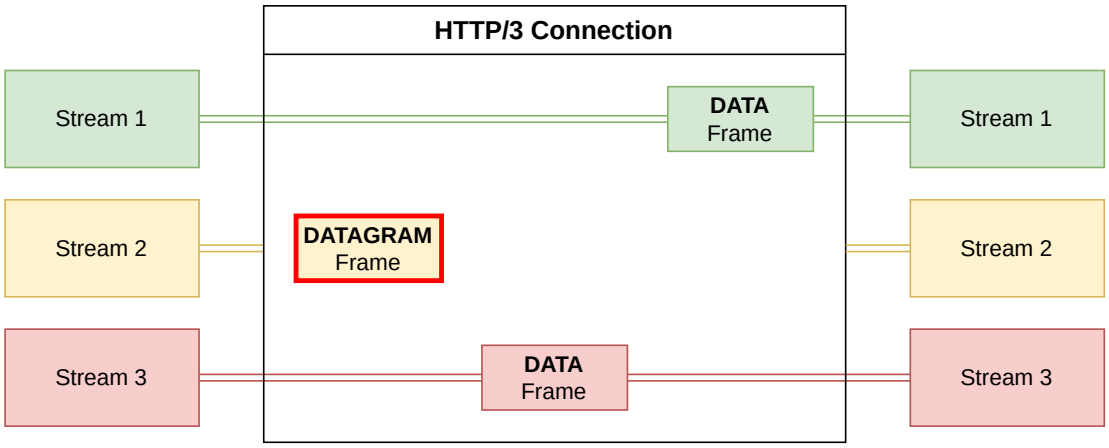

FIGURE 2.2: QUIC Streams Using Unreliable QUIC Datagrams

Key attributes of QUIC include:

- **Connection Establishment**: QUIC reduces the latency of connection setup by combining connection and transport handshake in Zero Round Trip Time (0-RTT).
- **Congestion Control (CC)**: Built-in CC mechanisms prevent network congestion, adapting to changing network conditions and optimizing data flow.
- **Security**: With QUIC, security features are intrinsic. It incorporates TLS 1.3, ensuring encrypted communication, authenticated connection establishment, and protection against various attacks.
- **Stream Multiplexing**: Unlike TCP, which uses multiple connections to handle concurrent data streams, QUIC can manage multiple streams within a single connection. This eliminates the need for multiple setups and teardowns of connections.
- **Flow Control**: QUIC offers both connection-level and stream-level flow control, ensuring efficient utilization of available network resources.

Figure 2.2 demonstrates how QUIC allows for the separation of multiple streams not only logically but also on a connection level, which helps prevent HOL blocking. Each stream can be transmitted independently without affecting the others. If a stream has to recover packets, the other streams are not affected.
It is worth noting that stream 2 in the figure uses the QUIC datagram extension [14]. This extension allows QUIC to carry datagram payloads that do not require the reliability guarantees of default QUIC packets. As a result, data sent within a QUIC

datagram is not bound to the properties of a stream. QUIC datagrams are critical to the implementation of MASQUE, which is discussed in subsequent chapters.

2.3 iCloud Private Relay

One practical application of MASQUE is found in iCloud Private Relay, a service provided by Apple that aims to enhance user privacy on the internet. This service employs a dual-relay system and various encryption mechanisms to ensure that no single entity has complete visibility of both a user's identity and their browsing activity. [15]

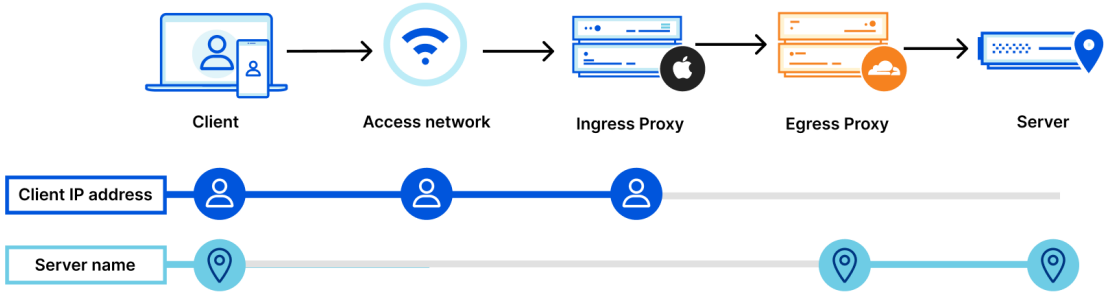

Figure 2.3: Diagram of the iCloud Private Relay System [16]

As depicted in Figure 2.3, the process begins with the user's device sending an encrypted request to a server, known as the *Ingress Proxy*, which is operated by Apple. This server can see the user's IP address but not the website they are trying to access, as this information is encrypted. The Ingress Proxy then passes the encrypted data to a second server, or *Egress Proxy*. This server cannot decipher the contents of the data or identify the user. Its role is to decrypt the website name and forward the user's request to the target server.

iCloud Private Relay utilizes several modern encryption and transport protocols, including TLS 1.3 and QUIC, to ensure secure and efficient data transfer. Notably, it incorporates MASQUE, which theoretically enables efficient data transfer between the multiple relay points involved in the process. This usage of MASQUE aims to ensure user privacy while minimizing the impact on browsing performance.

iCloud Private Relay serves as an example of a large-scale implementation utilizing the MASQUE protocol, illustrating its potential for enhancing user privacy and improving data transfer efficiency. The integration of MASQUE in iCloud Private Relay demonstrates the protocol's potential to address the challenges of secure and efficient data transmission in real-world scenarios.

Chapter 3

Related Work

It is worth mentioning that the MASQUE protocol is still in its early stages, and research and implementation efforts are currently limited. The existing implementations and studies are few, indicating that there is still much room for exploration and development. While only three implementation approaches, with only one covering CONNECT-IP, and a single paper on CONNECT-UDP's performance, have been identified, it is apparent that MASQUE is at the beginning of its journey.

As the protocol evolves, it is expected that additional implementations, studies, and advancements will emerge, contributing to a deeper understanding of MASQUE's capabilities and potential applications.

3.1 Head-of-Line Blocking

The HOL blocking problem, as discussed by Marx [17], is a significant issue in web communication. Essentially, when a single packet's delay impedes the processing of subsequent packets, it can introduce unwanted latency, degrading the performance of web applications. Marx traces this problem's lineage across various HTTP versions.

Historically, HOL blocking was a challenge in HTTP/1.1 due to its inability to efficiently multiplex responses. HTTP/2 aimed to tackle this by introducing frames and streams for resource chunk identification. However, the underlying TCP protocol still presented obstacles, resulting in HOL blocking when packets faced delays or losses. This limitation of TCP formed the foundation for the development of HTTP/3 and QUIC. With QUIC incorporating the principles of HTTP/2 at the Transport layer, the challenges posed by

TCP were addressed, providing a potential solution to the persistent problem of HOL blocking.

3.2 Technical Evaluation of QUIC and MASQUE

Kühlewind et al. [18] offer a comprehensive analysis of the effects of a MASQUE-based tunnel setup on end-to-end QUIC performance. Their research, based on the aioquic[1] QUIC and HTTP/3 stack, incorporates a Docker-based emulation environment for various network scenarios. In order to improve the aioquic HTTP/3 framework, they included support for HTTP datagrams and made adjustments to the QUIC system. The results, while indicative of potential advantages, also showcased certain limitations in the underlying stack and the simplicity of the emulation setup. These insights may be instrumental for future endeavors, especially for those focusing on MASQUE's extended performance metrics.

Scharnitzky et al. [19] took a different route. They developed a net device within the ns-3 open-source network simulator[2], aiming to emulate Long Term Evolution (LTE) networks in real-time. Their work focuses on enabling clients to communicate via an emulated LTE network, removing the need for specialized radio equipment. While the primary goal was the emulation of LTE and not MASQUE proxy performance, their differentiation between reliable and unreliable packet transmission offered crucial insights. They observed that in low-performance environments, the use of unreliable datagrams for MASQUE could potentially decrease the average completion time.

3.3 Practical Implementations and Implications

In a more specialized context, Markus Kraft [20] explored the advantages of MASQUE for enhancing satellite link performance, especially those with high round trip times. His tests pitted MASQUE against a VPN approach with comparable topologies. The results showed promise, with explicit proxy technologies potentially enhancing throughput over satellite links, even in scenarios with local network packet losses.

Cloudflare, a major player in the web technology arena, has its own take on proxying. As Galicer and Wood [21] explain, Cloudflare's Privacy Gateway focuses on ensuring user privacy. Based on the Oblivious HTTP (OHTTP) protocol, it acts as a mix network,

[1] `https://github.com/aiortc/aioquic`

[2] `https://www.nsnam.org`

concealing the source and destination of each message. While performance optimization is not the primary objective of the Privacy Gateway, it is essential to understand its place within the broader context of web proxying solutions. Given the thesis's focus on MASQUE proxying's performance aspects, this method will not be delved into further.

Chapter 4

MASQUE Proxying

With QUIC's capabilities such as separate streams and secure communication, MASQUE strives to tackle the obstacles that come with traditional proxying methods. In this section, we will explore the critical elements of the MASQUE protocol, the techniques it utilizes to enhance proxying, and the possible scenarios where it can be useful.

4.1 Overview

MASQUE leverages QUIC's ability to handle multiple independent streams concurrently. One of the key aspects of the MASQUE protocol is its use of the QUIC datagram extension, which enables the transmission of unreliable datagram payloads. These payloads are not subject to the same reliability guarantees as default QUIC packets, providing an opportunity for the proxied traffic to manage its own reliability requirements. This allows applications using the MASQUE protocol to benefit from the inherent reliability mechanisms built into the original traffic, instead of relying on the transport layer for this purpose.

In the MASQUE protocols, data (both UDP and IP) is encapsulated within QUIC unreliable datagrams. This encapsulation process involves taking the original data packets and embedding them as payloads within the datagrams. Once encapsulated, these datagrams are sent over QUIC connections to the proxy server, which then decapsulates the payloads and forwards the original data packets to their intended destinations.

The MASQUE protocols, mainly HTTP CONNECT-UDP and HTTP CONNECT-IP, extend the HTTP CONNECT mechanism to support different types of data transmis-

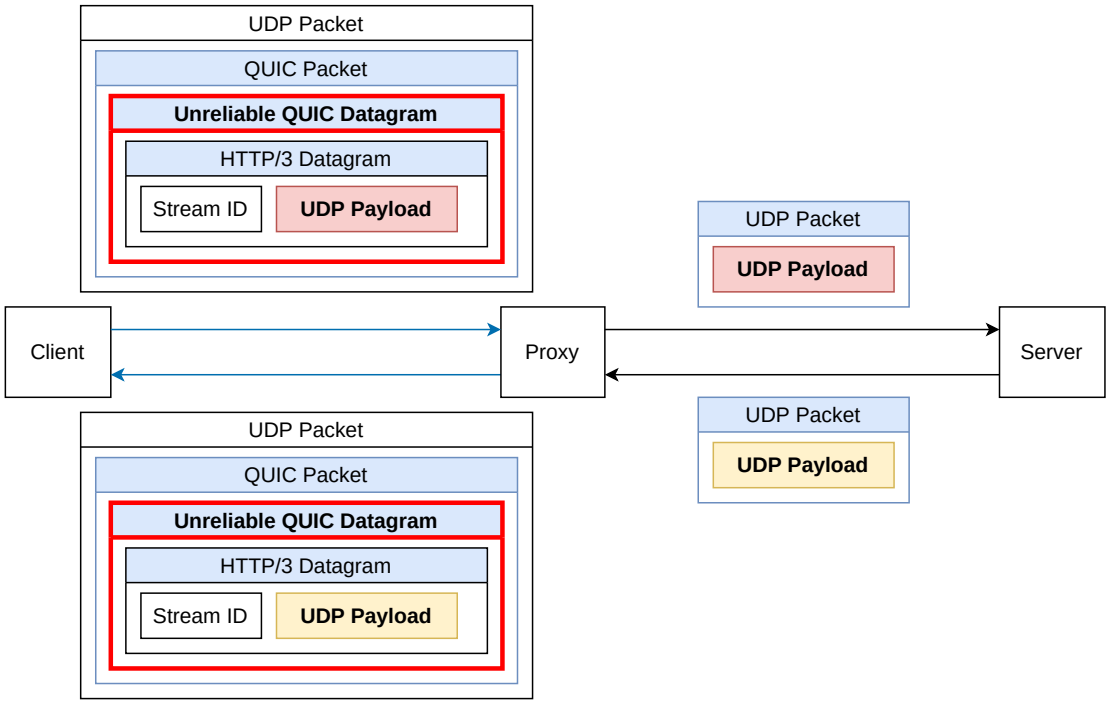

Figure 4.1: Connect-UDP

sion. Furthermore, it allows each data stream to be transmitted independently and in parallel, offering a more flexible approach to proxying.

4.2 HTTP CONNECT-UDP

Figure 4.1 illustrates the process of CONNECT-UDP proxying. In this scenario, the client wishes to send a UDP payload to the target server through the proxy.

1. The client first establishes a QUIC connection with the proxy and sends an HTTP request containing the CONNECT-UDP method, specifying the target server's address and port. This step informs the proxy about the intended destination for the UDP traffic.
2. The client then encapsulates the original UDP payload within a QUIC datagram, which is further encapsulated in a QUIC packet.
3. The QUIC packet, now containing the original UDP payload within the QUIC datagram, is sent to the proxy over the established QUIC connection.
4. Upon receiving the QUIC packet, the proxy extracts the UDP payload from the encapsulated QUIC datagram.

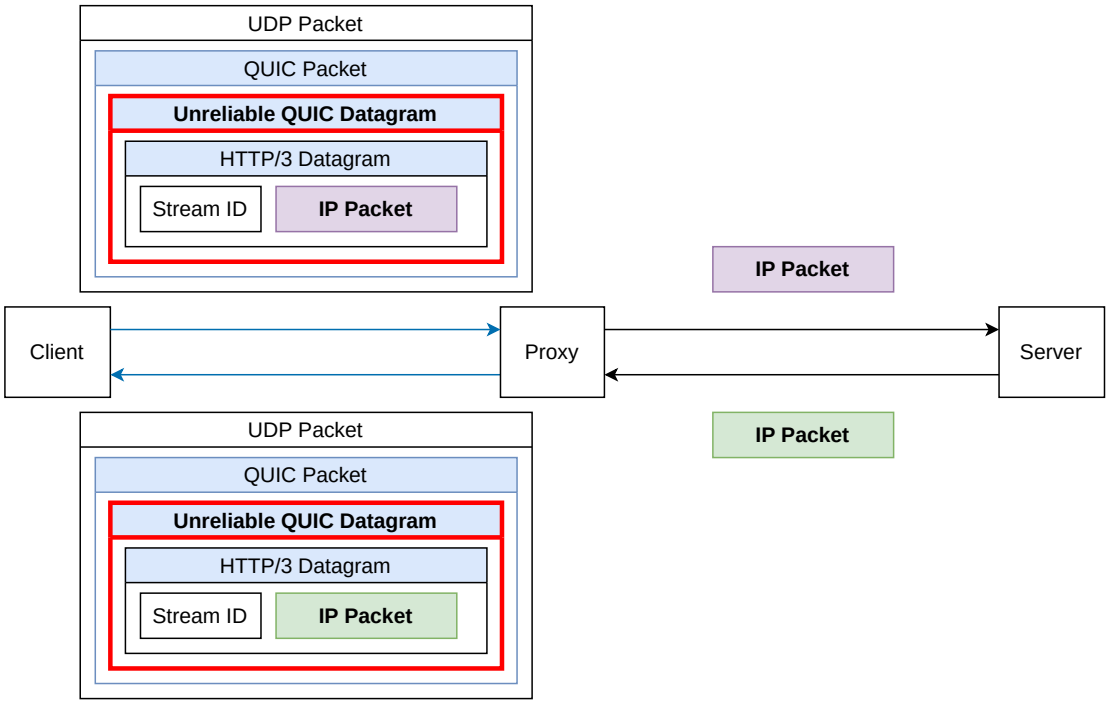

Figure 4.2: CONNECT-IP Proxying

5. The proxy forwards the extracted UDP payload to the target server, sending it in its name.

This process enables the client to transmit UDP traffic to the target server through the proxy. It should be noted that one CONNECT-UDP connection is limited to a single target server, as the target information must be exchanged beforehand between the client and the proxy.

4.3 HTTP CONNECT-IP

Figure 4.2 illustrates the process of CONNECT-IP proxying. In this scenario, entire IP packets are encapsulated within QUIC datagrams, which in turn are transmitted within QUIC packets.

In contrast to CONNECT-UDP, CONNECT-IP does not require specifying the target server beforehand, as the encapsulated IP packets contain destination addresses. This enables the proxy to forward the IP packets to the appropriate target servers based on the information included within the packets. As a result, one CONNECT-IP connection can be used to communicate with different targets, providing greater flexibility in proxying scenarios.

4.4 Advantages and Use Cases

MASQUE offers several theoretic advantages over traditional proxying methods, including enhanced performance, security, and versatility. By leveraging the strengths of QUIC and HTTP/3, it enables independent streams to be transmitted in parallel, eliminating the HOL blocking issue commonly encountered with TCP-based proxying. This leads to improved efficiency and reduced latency in network communications.

Some of the primary use cases for the MASQUE protocol include:

1. Proxying of UDP-based protocols: With HTTP CONNECT-UDP, the MASQUE protocol can efficiently proxy UDP-based traffic .
2. Generalized IP proxying: HTTP CONNECT-IP allows for the proxying of any IP-based traffic, supporting a wide range of applications and network configurations.
3. Improved performance in resource-constrained environments: By reducing latency and overcoming HOL blocking issues, the MASQUE protocol is particularly beneficial in environments with limited bandwidth or high-latency connections.
4. Enhanced security: By leveraging the encryption and authentication features of QUIC and HTTP/3, MASQUE provides a more secure proxying solution than traditional TCP-based methods.

Chapter 5

Analysis / Design

In this chapter, we will discuss the design of our MASQUE implementation, focusing on the motivation behind its development and the choice of libraries used.

5.1 Choice of Libraries

For our custom implementation, we chose to use Facebook's proxygen[1] and mvfst[2] libraries. Proxygen is a core C++ HTTP library used at Facebook and supports HTTP/1.1, HTTP/2, and HTTP/3. It provides support for the HTTP/3 datagram protocol, which is necessary for our implementation. Proxygen itself is based on mvfst, Facebook's implementation of the QUIC transport layer. Mvfst additionally provides support for the QUIC unreliable datagram extension.

It should be noted that the current version of proxygen does not offer support for MASQUE proxying. Although there is an exemplary client that mentions[3] support for CONNECT-UDP, it lacks specific protocol specification logic.

To implement the CONNECT-UDP and CONNECT-IP protocol, we make use of the existing QUIC/HTTP functionality of mvfst and proxygen, respectively, for the following reasons:

---

[1] `https://github.com/facebook/proxygen`

[2] `https://github.com/facebookincubator/mvfst`

[3] `https://github.com/facebook/proxygen/blob/ae8b4ce8/proxygen/httpclient/samples/H3Datagram/H3DatagramClient.cpp#L34`

1. Proxygen and mvfst are established libraries that have been designed to work together. By using these libraries, we can leverage their features and capabilities to develop a custom MASQUE implementation that meets our requirements and enables us to evaluate the protocol's performance and efficiency. The compatibility and design of these libraries make them a suitable choice for our implementation. Additionally, both libraries have friendly and well-documented interfaces.
2. By extending Facebook's libraries, we may test and evaluate our implementation against Google's QUICHE, which could give us more insight into the overall behavior of the MASQUE protocol.[1]

Using proxygen and mvfst for our MASQUE implementation enables us to analyse the protocols, and provide us with valuable insights into its behavior and potential for further development.

5.1.1 Comparison of QUIC Libraries

A detailed study of different QUIC implementations shows a variety in their underlying behaviors, and suggests that many of these libraries still need more optimization [22]. The following comparison is based on [23] and [22].

The QUIC libraries considered in this section include QUICHE[2] by Google, quiche[3] by Cloudflare, mvfst[4] by Facebook, MsQuic[5] by Microsoft, quic-go[6], aioquic[7], and haskell-quic[8]. The key features and differences of these libraries are summarized in Table 5.1.

Comparing these libraries, we can see significant diversity in the choices made for congestion control. In particular, mvfst stands out due to its extensive usage within Facebook's ecosystem, including their social networking sites Facebook and Instagram. It includes both NewReno and CUBIC congestion control algorithms, along with other options. This combination of broad usage and the presence of multiple congestion con-

[1] Note that this could not be explored further since QUICHE has not yet implemented full support for MASQUE: `https://github.com/google/quiche/tree/main/quiche/quic/masque`.

[2] `https://github.com/google/quiche`

[3] `https://github.com/cloudflare/quiche`

[4] `https://github.com/facebookincubator/mvfst`

[5] `https://github.com/microsoft/msquic`

[6] `https://github.com/quic-go/quic-go`

[7] `https://github.com/aiortc/aioquic`

[8] `https://github.com/kazu-yamamoto/quic`

Table 5.1: Comparison of different QUIC Libraries

Library	Language	Congestion Control
QUICHE	C++	BBR, CUBIC
quiche	Rust	BBR, CUBIC, Reno
mvfst	C++	BBR, COPA, CUBIC, NewReno, StaticCwnd
MsQuic	C/C++	BBR, CUBIC
quic-go	Go	CUBIC, HyStart
aioquic	Python	NewReno
haskell-quic	Haskell	NewReno

Table 5.2: Mean goodput of different QUIC Libraries [24]

Library	Goodput (Mbit/s)	Language	Developed by
quiche	2006	Rust	Cloudflare
mvfst	891	C++	Facebook
lsquic	857	C	LightSpeed Tech
aioquic	55	Python	aiortc

trol algorithms positions mvfst as a feature-rich QUIC implementation. However, it should be noted that many QUIC libraries, including mvfst, have ongoing work regarding their congestion control implementations and have not been extensively validated for performance or fairness [22].

5.1.2 Performance of QUIC Libraries

An important aspect to consider in addition to the features provided by these QUIC libraries is their performance. Various studies provide detailed insights into the performance of different QUIC implementations, especially focusing on high-rate links [24], [25].

The goodput of different QUIC libraries as measured on a 10Gbit host pair with 5 repetitions and a 10GB filesize is summarized in Table 5.2.
In addition to goodput, [24] also analyzed the breakdown of CPU usage for different QUIC implementations, including quant[1], quicly[2], picoquic[3], and mvfst, all written in C or C++. In particular, mvfst showed a throughput of around 300 Mbit/s with a CPU

[1] `https://github.com/NTAP/quant`

[2] `https://github.com/h2o/quicly`

[3] `https://github.com/private-octopus/picoquic`

usage of cryptography functions of about 10% to 13% on the server. The packet I/O CPU usage was dominated by Cloudflare's quiche, both on the client and the server.

[25] further evaluated the performance of these QUIC implementations on high-rate links. Their findings suggest that goodput performance varies significantly not only between different QUIC libraries, but also between client-server pairs. Libraries such as LSQUIC[1] and quiche showed the highest goodput, with goodput rates varying from 52 Mbit/s to 3004 Mbit/s depending on the client-server combination. The results also revealed that QUIC implementations can be greatly impacted by the operating system parameters, such as the default buffer size, and hardware features, such as Network Interface Card (NIC) offloading and Advanced Encryption Standard New Instructions (AES NI) hardware acceleration.

In terms of performance, while mvfst has a lower goodput compared to some other libraries, it is still competitive considering its broad usage and robust feature set. Future work could focus on optimizing the performance of mvfst to further improve its goodput, CPU usage, and adaptability to various operating systems and hardware configurations.

[1] `https://github.com/litespeedtech/lsquic`

Chapter 6

Implementation

This chapter provides an overview of our MASQUE implementation, outlines the challenges we faced, and presents the solutions we developed.

6.1 Server

We based our server on Proxygen's HQServer[1], a sample implementation for an HTTP/3 server.

[1] `https://github.com/facebook/proxygen/blob/main/proxygen/httpserver/samples/hq/HQServer.cpp`

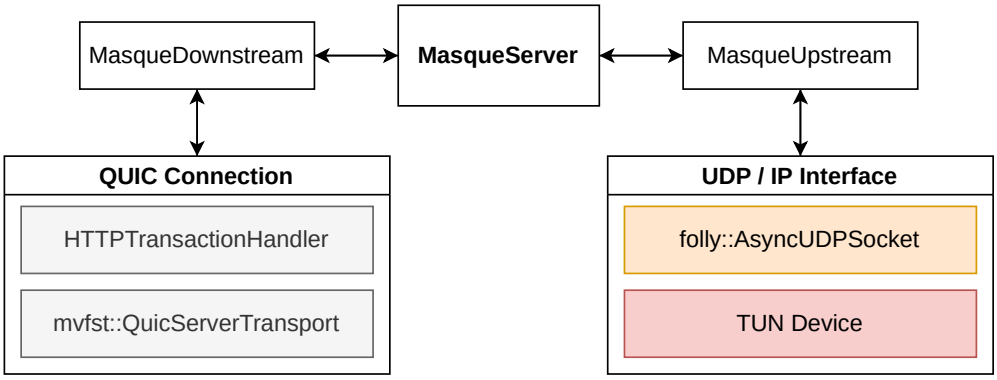

Figure 6.1: Architectural Overview

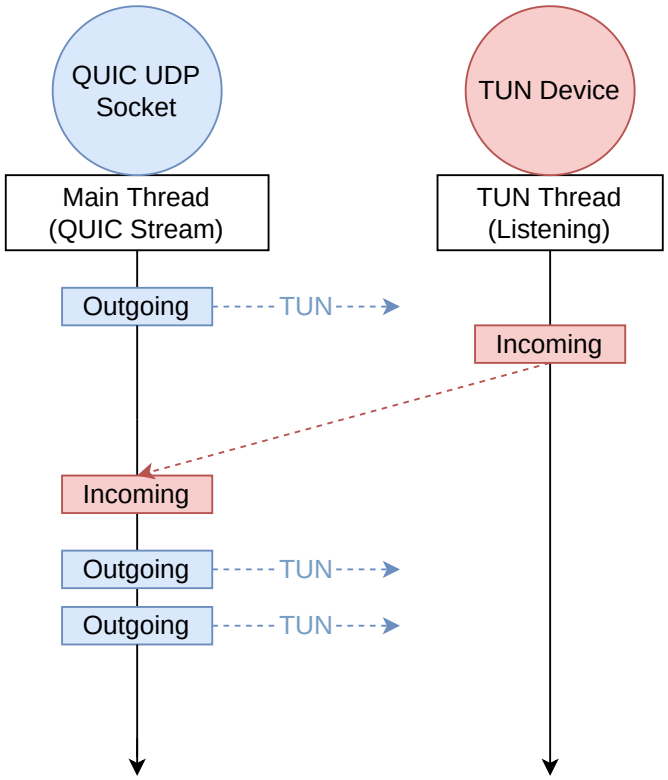

Figure 6.2: Proxygen Multithreading Overview for CONNECT-IP

In order for our proxy server to send and receive IP packets, we make use of a TUNnel (TUN) device[1]. TUN devices are virtual networks which allow user space programs to write and receive packets on the network layer. Using `read()` and `write()` on a TUN's file descriptor, the server can handle arbitrary IP traffic.

As visualized in Figure 6.1, for each successful incoming MASQUE request, we

- bind a new `AsyncUDPSocket` to the respective target server (for CONNECT-UDP).
- register a new IP within our TUN subnet and bind it to the stream (for CONNECT-IP). The IP will then be mapped to the stream ID upon receiving incoming packets.

Thus, the server only has to maintain one TUN device for all streams.

The `AsyncUDPSocket` binds immediately to the target server from the request, and for the TUN device, we instantly send the subnet or the available IP range for the client to the client through the `ADDRESS_ASSIGN` capsule. This process allows the client to use

[1] `https://www.kernel.org/doc/html/v6.6/networking/tuntap.html`

a real IP from the TUN subnet as the source IP for its packets, which eliminates the need for manual packet rewriting.

Upon receiving an IP packet (CONNECT-IP) or a UDP payload (CONNECT-UDP), we forward it to the TUN device or respective socket. The TUN device or `AsyncUDPSocket` then communicates with the target server. We send any responses back to the client via the unreliable datagram stream.

The TUN device and all of the UDP sockets run in their own thread, internally using batched operations with `io_uring`[1] for IO. This allows us to minimize the number of syscalls per IO operation, keeping the overall overhead of the TUN device as small as possible. Upon receiving an IP packet from the client within a datagram, the main stream hands it to the TUN thread, which buffers it until the TUN device is ready to write again (as depicted in Figure 6.2).

This method enables us to run a single server supporting both CONNECT-UDP and CONNECT-IP connections from multiple clients.

6.2 Client

In this section, we focus on the client-side implementation, outlining the design and function of the TUN client and the methodology behind managing multiple hops.

6.2.1 TUN Client

Our primary client interacts with a TUN device for data input.

- For CONNECT-IP: After receiving the `ADDRESS_ASSIGN` capsule, the client initiates the TUN device and configures it with the respective received subnet. When this TUN device acquires an IP packet, we encapsulate it into a datagram and forward it directly to the proxy server. The same process applies to receiving data. Figure 6.3 illustrates this workflow.
- For CONNECT-UDP: We follow the same process, but first, we parse the packet, i.e., extract the UDP payload before sending it via CONNECT-UDP. As a result, this method only supports UDP traffic (as expected).

Consequently, a client may only run in either CONNECT-IP or CONNECT-UDP mode. Note that the client also supports opening multiple HTTP/3 transactions on the same

[1] `https://www.phoronix.com/news/Linux-io_uring-Fast-Efficient`

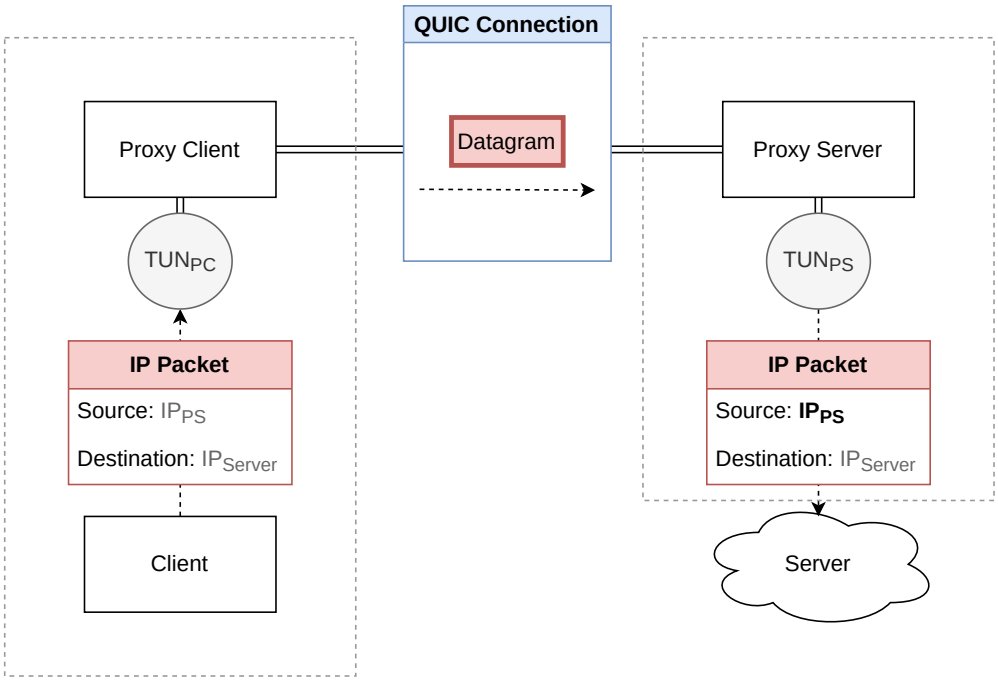

Figure 6.3: CONNECT-IP Implementation Concept

QUIC connection. For each new transaction, the client spawns a new TUN device which may then be used exclusively by the respective process.

6.2.2 Multiple Hops

To route our traffic over multiple MASQUE proxy servers, our client must support layered connections. As shown in Figure 6.4, we have implement this feature through `LayeredConnectIPSocket` and `LayeredConnectUDPSocket`, both of which inherit from the `AsyncUDPSocket` interface. While a `LayeredConnectUDPSocket` may just forward the UDP payload, a `LayeredConnectIPSocket` first has to encapsulate it in an UDP/IP packet since the proxy server expects a valid IP packet to forward.

Internally, both use proxygen's `H3DatagramAsyncSocket`[1], which also inherits from `AsyncUDPSocket`. This design allows us to pass a `LayeredConnectIPSocket` or `LayeredConnectUDPSocket` as a base layer to a `H3DatagramAsyncSocket`, effectively enabling us to layer traffic.

[1] `https://github.com/facebook/proxygen/blob/main/proxygen/lib/transport/H3DatagramAsyncSocket.cpp`

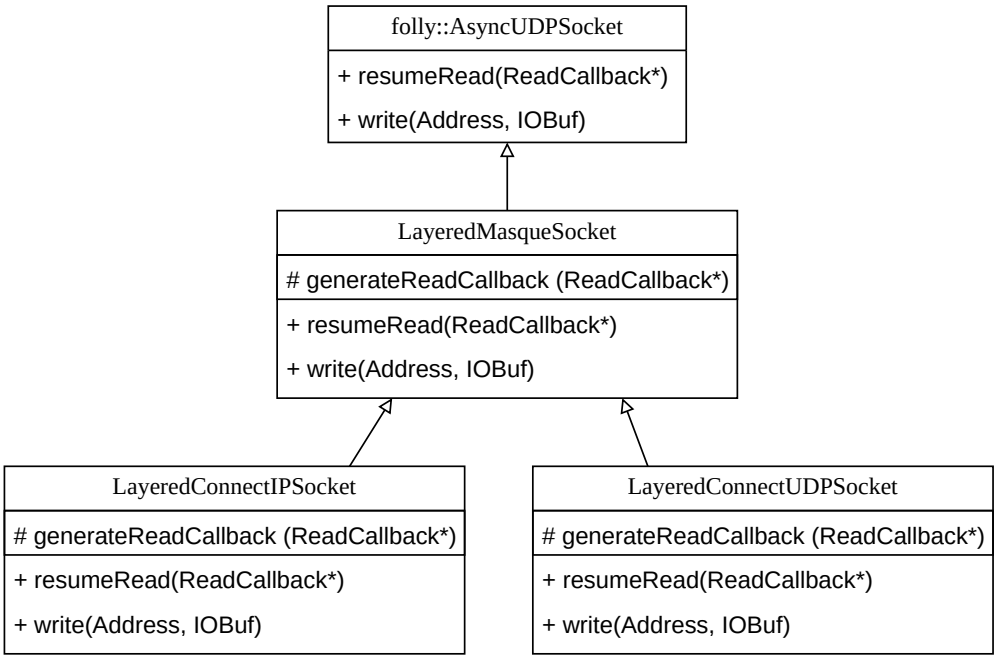

Figure 6.4: Layered TUN Client

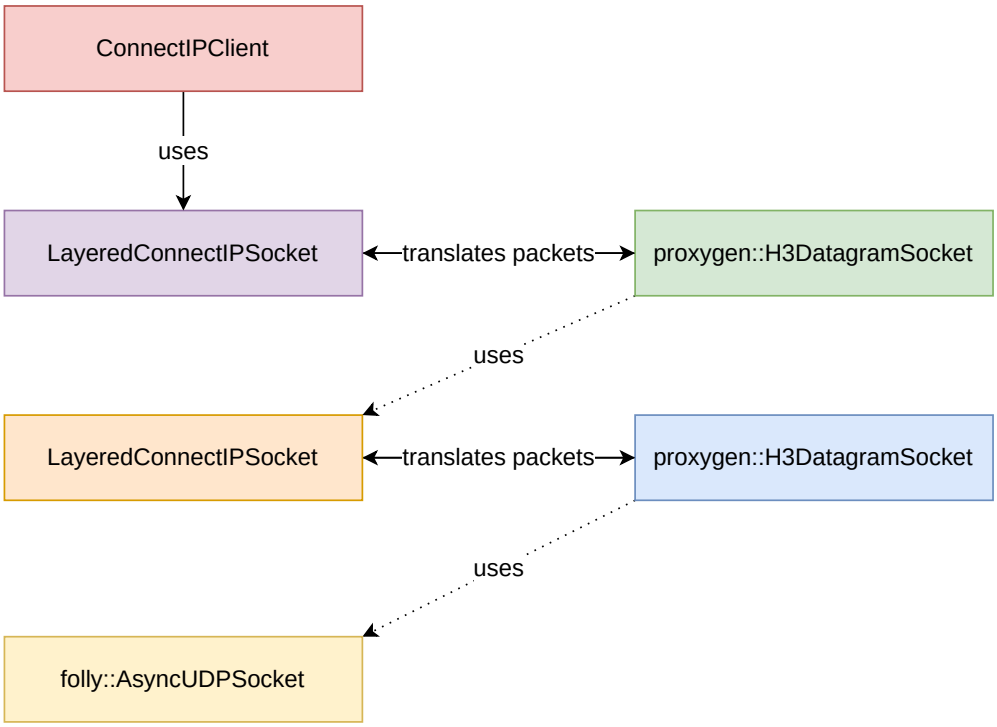

Figure 6.5: TUN Client Hierarchy

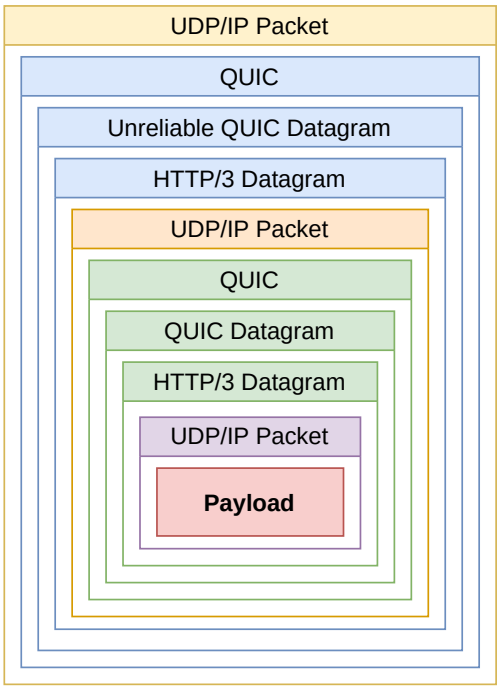

Figure 6.6: Layered CONNECT-IP Packet

Figure 6.5 illustrates a MASQUE connection via two CONNECT-IP proxy hops, showing a hierarchy of `LayeredConnectIPSockets` that utilize `H3DatagramSockets`, which, in return, employ the next `LayeredConnectIPSocket`, and so forth. This structure allows for layered, multi-hop connectivity.

Figure 6.6 shows our implementation of the layered CONNECT-IP approach through manual packet crafting. Given that each socket interface expects a payload passed to its `write()` method, we create a new UDP/IP packet with fixed ports for each new hop. These manually crafted packets serve as the payload in our datagrams. Upon receiving the layered datagram, the proxy server can then extract the encapsulated UDP/IP packet to forward it via its TUN device. The layers of a layered CONNECT-IP packet in Figure 6.6 are color-coded to correspond with the instances in the client depicted in Figure 6.5. Each packet part is colored to indicate at which layer it is added to the overall structure, making it easier to understand the multi-hop packet crafting process.

In the MASQUE protocol, the overhead for each CONNECT-IP packet plays a key role for the effective throughput. This overhead consists of various components, as outlined in Table 6.1. Given a standard Maximum Transmission Unit (MTU) of 1500B and considering the 1200B minimum packet size required by the MASQUE RFC, a careful

Table 6.1: Detailed Breakdown of the Packet Sizes for MASQUE CONNECT-IP.

Component	**Size Bounds (B)**	**Actual Sizes (B)**
Masque Context ID	1 - 8	1
HTTP/3 Datagram Frame Header	1 - 8	8
QUIC Datagram Frame Header	1 - 9	8
QUIC (short) Header	2 - 25	25
QUIC MAC	16	16
UDP Header	8	8
IPv4 Header	20	20
Total Overhead Per Hop	49 - 94	86

estimation allows for up to three hops. Yet, our practical findings indicate that it is feasible to achieve four hops without exceeding the MTU limits.

Note that, given the default MTU of 1500B, we cannot use more than two hops to proxy arbitrary QUIC traffic itself since the standard assumes a minimum MTU of 1280B [12], which CONNECT-IP barely falls short of after two hops.

6.2.3 Multiple HTTP/3 Streams over QUIC

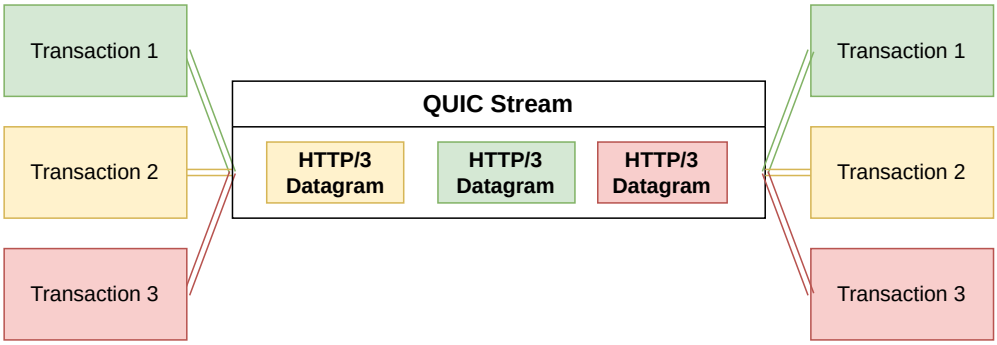

Figure 6.7: Multiple HTTP/3 Transactions Sharing one QUIC Stream

Through proxygen's architecture, our system additionally establishes multiple HTTP/3 transactions concurrently on a single QUIC connection. Figure 6.7 provides a visual representation of multiple transactions over one QUIC connection. Each transaction possesses a distinct bidirectional stream identifier, ensuring a clear separation of different HTTP/3 frames and thus preventing any potential data overlap.

As outlined in the HTTP/3 documentation RFC 9114 [13], the protocol supports a robust capacity for multiple streams. QUIC itself oversees the management of these concurrent streams.

When comparing HTTP/2 and HTTP/3, the latter seems better equipped to handle more client-initiated bidirectional streams simultaneously. This design is intended to make HTTP/3's concurrency similar to HTTP/2, based on expected user activities. Importantly, the way QUIC manages these streams provides a solution to the Head-of-Line blocking issue, further explained in section 2.2.

6.2.4 HTTP Client

To complement our setup and perform testing without the overhead of the client TUN, we additionally implemented a simple HTTP client supporting GET requests.

By default, the client is configured to download a 1GB file. Importantly, the client also offers flexibility to split this download over multiple concurrent HTTP transactions. This feature aligns with our goal to test and evaluate the system's capability to handle parallel transactions.

For our experiments, we use the integrated `hq-server`[1] of proxygen, a sample HTTP/3 server implementation. This server supports multiple HTTP transactions over one QUIC connection, making it ideal for out purposes.

6.3 Logging and Metrics

To ensure comprehensive monitoring and performance analysis, our implementation integrates logging capabilities at various levels.

6.3.1 QUIC qlog Format

Utilizing the `mvfst` library, we support logging in the QUIC qlog [26] format. This standardized logging format offers in-depth inspection and debugging of the QUIC protocol behaviors. The qlog format captures information on packet transmission, reception, and other key QUIC events, offering high-level visibility into the QUIC state machine.

qlog provides the essential data required for automated metrics calculation. This eliminates the need for manual packet extraction and decryption via methods such as the combination of `SSLKEYLOGFILE` and `tcpdump`. Through this approach, we can compute key performance metrics, such as latency, throughput, and error rates, in an automated and streamlined manner.

[1] `https://github.com/facebook/proxygen/tree/main/proxygen/httpserver/samples/hq`

We implemented this logging functionality in the `hq-server` and our MASQUE HTTP client. This allows us to capture and map corresponding HTTP/3 and MASQUE-related events on both sides.

6.4 Challenges and Solutions

6.4.1 Faced Problems

During our work on implementing MASQUE proxying with proxygen and mvfst, we ran into several challenges. These are briefly outlined below:

- *Outdated CONNECT-UDP Draft*: mvfst seemed to partly consider an older version of the CONNECT-UDP draft, specifically regarding context-ids of HTTP/3 datagrams, which were first defined in [27]. We fixed this by updating mvfst to include the correct context-id, defined as zero.
- *maxFrameSize Parameter Issue*: The 'maxFrameSize' in mvfst was defined as a `uint16_t`, which overflowed at 2^{16}. We fixed this by changing it to a variable-sized integer.
- *QUIC Version Number Mismatch*: mvfst uses custom QUIC version numbers, specifically
  - `QUIC_DRAFT = 0xff00001d`
  - `QUIC_V1 = 0x00000001`
  - `QUIC_V1_ALIAS = 0xfaceb003`

  However, the current QUICv2 draft defines the version field as `0x6b3343cf` [28]. We addressed this by aligning the mvfst QUIC version number with the draft.
- *Repeated QUIC Transport Parameters*: mvfst was sending the same QUIC transport parameters more than once, which caused the reference implementation to stop the connection. We fixed this by making sure these parameters are sent only once.
- *Non-standard HTTP/3 Protocol Identifier*: mvfst uses a custom HTTP/3 protocol identifier, "h3-fb-05", instead of the standard "h3" per default. This caused negotiation problems with the reference implementation. We solved this by changing the default identifier in mvfst to "h3".

- *Missing State HTTP Transitions*: `HTTPTransactionEgressSM.cpp`[1] in proxygen defines the states for an HTTP connection, such as when to expect headers, a body, and so forth. However, the immediate transition from headers to datagrams, as defined in CONNECT-UDP, was missing. We addressed this issue by adding the necessary state transitions.
- *Issues with libviface for TUN Device*: Initially, we attempted to use libviface[2], but encountered problems due to its use of multiple queues[3]. libviface separates the read and write operations for the TUN into two different file descriptors. However, we found that the kernel was also sending packets to the write file descriptor, which wasn't being read from. We subsequently switched to libtuntap[4], which provided the same functionality. We directly used the C library instead of the C++ wrapper (libtuntap++) since we consider it unnecessary to use a thin wrapper that just forwards the calls.
- *Static Default Values in mvfst*: mvfst defines static default values for QUIC in `QuicConstants.h`[5]. This includes parameters such as `kDefaultUDPSendPacketLen` (default QUIC packet size for both read and write) and `kDefaultMaxUDPPayload` (the maximum for a peer's `max_packet_size`). However, because our layered proxying approach has to deal with varying sizes on each layer, it was necessary to make these parameters modifiable.
- *HTTP/3 Stream ID Integration in Proxygen*: While proxygen's original design hides the stream ID, our approach requires its direct association with datagrams. To meet this need, we've adapted proxygen to send data to a specific stream using its stream ID. This change allows multiple concurrent HTTP transactions over a single QUIC connection, each linked to its respective stream ID.
- *Issues with Concurrent HTTP Transactions in Quiche-Server*: For experiments involving multiple concurrent HTTP transactions, we initially tested using `quiche-server`[6] from Cloudflare's quiche. However, we found that `quiche-server`

[1] `https://github.com/facebook/proxygen/blob/main/proxygen/lib/http/session/HTTPTransactionEgressSM.cpp`

[2] `https://github.com/HPENetworking/libviface`

[3] `https://github.com/HPENetworking/libviface/issues/6`

[4] `https://github.com/LaKabane/libtuntap`

[5] `https://github.com/facebookincubator/mvfst/blob/main/quic/QuicConstants.h`

[6] `https://github.com/cloudflare/quiche/blob/master/apps/src/bin/quiche-server.rs`

does not fully support parallel transactions when interacting with mvfst. Specifically, `quiche-server` failed to immediately recognize and respond when more than one stream was active. We therefore switched to the sample HTTP/3 server implementation of proxygen.

Furthermore, in some instances, the behaviour of the TUN device on the nodes differed from observations during local developement: Although we did not change the default kernel log level of WARNING (4), the TUN device issued an informational message[1] (log level 6) to `/dev/kmsg` each time, we sent a packet through it. Thus, a lot of CPU time was occupied for dealing with the logs, lowering the throughput results.
We tackled this problem by explicitely setting the log level in `/proc/sys/kernel/printk` before running the experiments, which seems to fix the issue.

Navigating through these challenges, we successfully integrated MASQUE proxying. This experience highlights the critical need to regularly update libraries and closely follow protocol standards. It also emphasizes the care required when merging different software elements.

[1] `https://elixir.bootlin.com/linux/latest/source/drivers/net/tun.c#L1091`

Chapter 7

Evaluation

This chapter presents the evaluation and analysis of the MASQUE implementation and its performance in various scenarios, including library-specific comparisons. The goal of these experiments is to address the research questions outlined in section 1.2.

7.1 Setup Overview and Methodology

This section elaborates on the configurations used to test the efficiency and behaviour of our MASQUE implementation. Our experimental framework includes a pool of nine servers: bitcoin, bitcoincash, bitcoingold, dogecoin, dogecoincash, dogecoingold, ether, ethercash, and ethergold. Each server[1] connects through a 1Gbit/s link on the `eno5` interface (full duplex mode), coordinated by a switch (see Figure 7.1).

In addition to `eno5`, we also employ the connection chain via `eno3` and `eno4` as depicted in Figure 7.2, which allows us to rule out the single switch to be the bottleneck in certain scenarios. We use this configuration for all experiments if not specified otherwise.

7.1.1 Testbed Setup With Multiple Hops

The following example shows how we use the server topology for a multi-hop setup in which the client connects via more than one proxy servers to the respective target.

---

[1] OS: Debian 10 | CPU: Intel Xeon D-1518 (4 cores / 8 threads) | Memory: 32GB

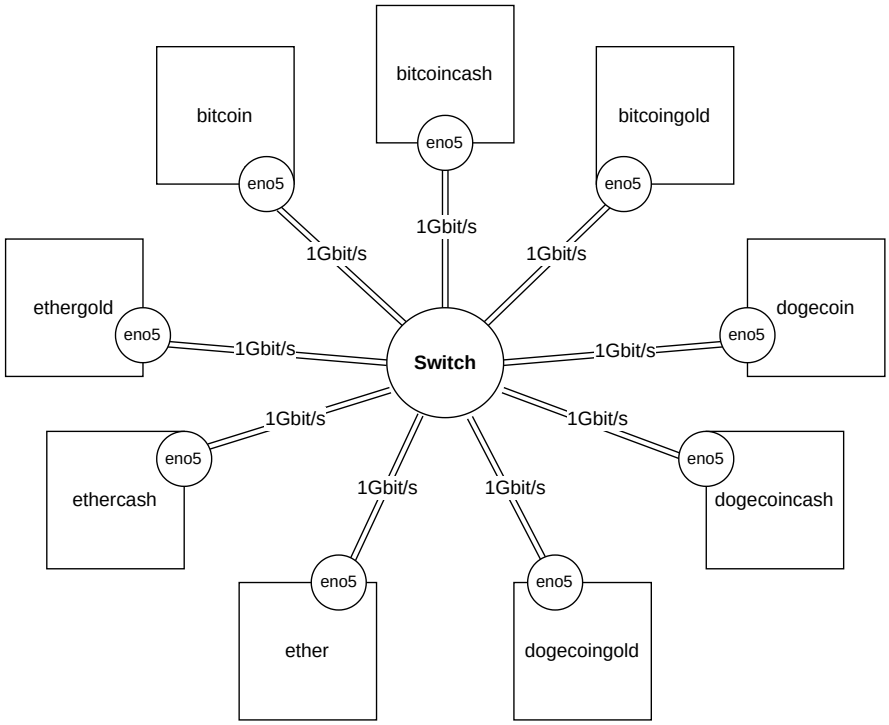

Figure 7.1: Server Topology (`eno5`)

Here, the client is placed on bitcoin. bitcoincash acts as the first proxy, followed by bitcoingold as the second proxy, and finally, ether stands as the target server (see Figure 7.3).

7.1.2 Setup Parameters and Metrics

For the evaluation and analysis, we consider the following metrics to measure the performance of our MASQUE implementation:

- **Throughput**: the rate of data transfer between the client and the target server.
- **Time To First Byte (TTFB)**: the time it takes for the first byte of data to arrive at the client after sending a request.
- **Round Trip Time (RTT)**: the time it takes for a packet to travel from the client to the server and back to the client.
- **Latency**: the total time it takes for a data packet to travel from the source to the destination.
- **Jitter**: the variability in latency over time in the network, reflecting the fluctuations in the delay of packet delivery.

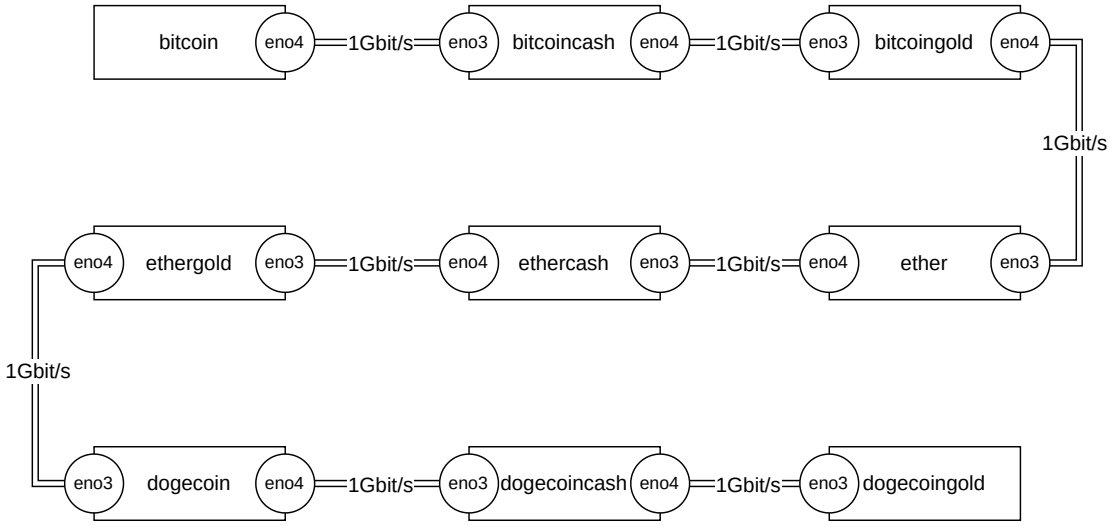

Figure 7.2: Server Topology (`eno3` + `eno4`)

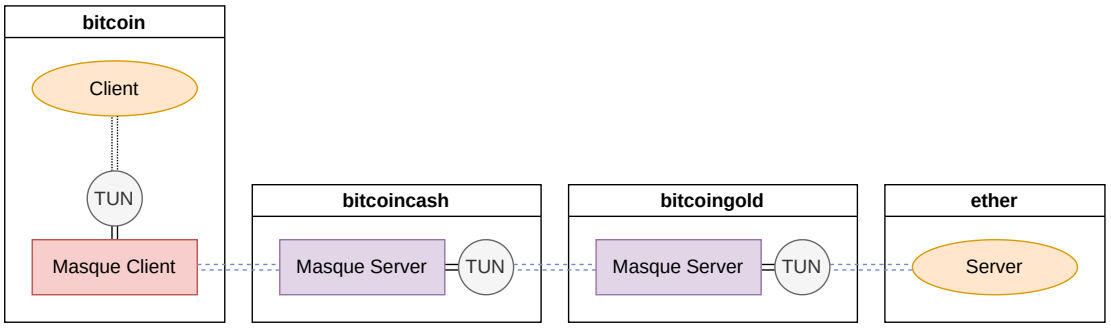

Figure 7.3: Two-Hop Sequence

We will also define network environment factors like bandwidth, packet loss, and RTT limits to ensure that the performance evaluation is comprehensive and realistic.

7.2 Experiments

For each of the following experiments, we describe the respective setup in detail, followed by challenges we faced.

If not specified otherwise, we pin each thread to one core using `pthread_setaffinity_np` in our client and proxies, which allows us to have an overview of each thread's individual CPU usage. For this, we keep a separate background process during the experiments that regularly logs the current CPU workload with `mpstat`.

Table 7.1: Breakdown of Server Roles for the TunConnectIP Experiment.

Server	**Role**	**Executed Program(s)**
bitcoin	client	`masque-client`, `iperf3`
bitcoincash	hop 1	`masque-server`
dogecoin	hop 2 / iperf server	`masque-server` / `iperf3`
dogecoincash	hop 3 / iperf server	`masque-server` / `iperf3`
ether	iperf server	`iperf3`

7.2.1 TunConnectIP

The TunConnectIP experiment has the main goal of demonstrating the behaviour of MASQUE proxying with external applications, i.e. binding them to a client TUN device.

For the TunConnectIP experiment, we first spawn $n \in \{1, 2, 3, 4\}$ proxy servers on different nodes, followed by a `masque-client` instance. The client uses CONNECT-IP for all of the nested hops, which are set up sequentially. We then open both an `iperf` server and client, bind the client to our TUN device[1], and perform the throughput measurements (see table 7.1).
Note, that `iperf3` does not allow us to repeat the same experiment with a layered CONNECT-UDP connection as it first requires a TCP connection for communication[2].

In addition to the number of hops, we also make use of the client's option to spawn multiple HTTP/3 transaction. For each transaction, we create a new iperf client-server pair on a new port.

7.2.2 HTTPConnectIP / HTTPConnectUDP

The HTTP experiments help us to understand how masque proxying itself behaves in an environment using HTTP/3. For this, we employ our HTTP client natively supporting MASQUE to eliminate as many additional variables as possible. We believe that this allows us to examine the behavior in the most isolated way. Therefore, in contrast to the TunConnectIP experiement, we include all of the metrics listed in 7.1.2.

For both the HTTPConnectIP experiment and HTTPConnectUDP, we first spawn $n_{ip} \in \{0, 1, 2, 3\}$ / $n_{udp} \in \{0, 1, 2, 3, 4\}$ hops on different nodes, followed by a `masque-http-client`

[1] This requires us to use `iperf3` with its `--bind-dev` option.

[2] `https://github.com/esnet/iperf/issues/1331#issuecomment-1120186827`

Table 7.2: Breakdown of Server Roles for the HTTPConnectIP Experiment

Server	Role	Executed Program(s)
bitcoin	client	`masque-http-client`
bitcoincash	hop 1 / http server	`masque-server` / `hq-server`
bitcoingold	hop 2 / http server	`masque-server` / `hq-server`
ether	hop 3 / http server	`masque-server` / `hq-server`
ethercash	http server	`hq-server`

Table 7.3: Breakdown of server roles for the HTTPConnectUDP experiment

Server	Role	Executed Program(s)
bitcoin	client	`masque-http-client`
bitcoincash	hop 1 / http server	`masque-server` / `hq-server`
bitcoingold	hop 2 / http server	`masque-server` / `hq-server`
ether	hop 3 / http server	`masque-server` / `hq-server`
ethercash	hop 4 / http server	`masque-server` / `hq-server`
ethergold	http server	`hq-server`

instance (see tables 7.2 and 7.3). The client then uses the hops to download a 1GB file served by the `hq-server` on each transaction[1].

Since our HTTP client itself incorporates both the functionality needed for MASQUE proxying and raw QUIC traffic, we can also abstain from proxying the traffic at all, giving us a clearer upper limit for the possible throughput.

7.2.3 SeleniumConnectIP

In the SeleniumConnectIP experiments, we utilize several techniques to make Google Chrome bind to our proxy setup and force it to use QUIC for connecting to our HTTP server, which hosts a mirror of the TUM website[2]. This setup aims to provide a realistic testing environment, with the entire page providing about 1.9MB of content[3].

Because we aim to get insight into the general end user experience, we focus on the page loading time as well as the TTFB in the context of the initial HTTP request.

[1] For > 32 transactions, we gradually lower the individual file size to lower the respective total experiment time.

[2] `https://www.tum.de/`

[3] We disable `mp4` files since Chrome has the habit of "streaming" the videos rather than immediately downloading them. This removes about 13MB from the total page size, which is not needed for the main functionality of the site.

The experiment is constrained to a maximum of two proxy hops due to QUIC's minimum MTU assumption of 1280B, as discussed in Chapter 6 (6.2.2). Chrome's HTTP/3 connection itself can be understood as the third encapsulated tunnel.

Each client connects via the `eno5` interface over the switch to the first proxy in parallel, which then connects through the `eno4-eno3` chain over the other proxy hops to the HTTP server.

Because we focus on user experience, we perform the SeleniumConnectIP experiments under emulated network conditions:

- 10% packet loss
- 200ms RTT
- 1Mbit/s bandwidth

These are realistic parameters, modeling a communication via Long Term Evolution for Machines (LTE-M) over 15km [29]. We enable them using `tc`[1] and Selenium's `set_network_conditions()`[2].

Key elements and considerations regarding the setup include:

1. **BindToInterface Library**: We use the BindToInterface library[3] to bind the Chrome instances to the respective client TUN devices. This library allows the specification of network interfaces for applications, ensuring that the traffic from the browser is routed through the designated proxy path.
2. **Chrome Configuration**: To enforce the use of QUIC, we configure Chrome with specific flags[4]. The `--enable-quic` flag is essential to activate QUIC protocol support. Additionally, we use the `--origin-to-force-quic-on` flag to specify the ip of our server, forcing QUIC usage for connections to this domain. The `--ignore-certificate-errors-spki-list` flag is used to handle certificate validation, allowing us to bypass errors related to self-signed or invalid certificates.

---

[1] `https://man7.org/linux/man-pages/man8/tc.8.html`

[2] `https://www.selenium.dev/selenium/docs/api/py/webdriver_chromium/selenium.webdriver.chromium.webdriver.html#selenium.webdriver.chromium.webdriver.ChromiumDriver.set_network_conditions`

[3] `https://github.com/JsBergbau/BindToInterface`

[4] `https://github.com/aiortc/aioquic/tree/main/examples#chromium-and-chrome-usage`

3. **TTFB Measurement**: While Google Chrome does not support qlog natively, opting for netlog[1] instead, we encountered limitations with this approach. Even though tools like qvis[2] can convert netlog to qlog, we found that sometimes netlog does not capture all QUIC packets. This inconsistency rendered the netlog to qlog conversion approach not feasible for our needs.
   For measuring TTFB, we employed `goog:loggingPrefs` within our Selenium script. This feature enables the collection of detailed network performance logs from the Chrome browser, allowing us to precisely measure the time intervals for network requests and responses.

4. **Page Loading**: Selenium itself does support a general way of ensuring that the entire page content has sucessfully been loaded. Usually, we use `driver.get()` to open a url. This, however, does neither guarantee that all files have been loaded, nor that the website could not have been used prior to the call returning (i.e. by interacting with a partially loaded site). Therefore, we first set the `pageLoadStrategy`[3] to *eager*, which lets `driver.get()` already return earlier upon DOM access. To check if all files have been loaded, we use a polling strategy of getting a list of all finished requests using `getEntries()`[4]. We then calculate the number of received bytes, aka the respective `transferSize`. If this number is equal to the size of hosted page files, the page has been loaded.
   Additionally, to prevent Chrome from loading files from cache, we use the DevTools protocol[5] to toggle `Network.setCacheDisabled`. This makes Chrome completely ignore any cached files upon new requests.

5. **Alternatives to Chrome**: Although we primarily used Chrome for this experiment, it is worth noting that Firefox could potentially be used as well by manually overriding the `alt-svc` header[6]. However this approach was not satisfactory since the QUIC handshake still did not always succeed.

---

[1] `https://www.chromium.org/developers/design-documents/network-stack/netlog/`

[2] `https://qvis.quictools.info/`

[3] `https://www.selenium.dev/documentation/webdriver/drivers/options/#pageloadstrategy`

[4] `https://developer.mozilla.org/en-US/docs/Web/API/Performance/getEntries`

[5] `https://chromedevtools.github.io/devtools-protocol`

[6] `https://blog.cloudflare.com/how-to-test-http-3-and-quic-with-firefox-nightly/`

By employing these methods, we are able to route Chrome's QUIC traffic through our proxy setup. This approach allows us to gather detailed data on QUIC's behavior in a experimental setting, especially focusing on TTFB as a critical metric.

However, it should be noted that using Chrome via Selenium instead of, e.g. our HTTP client, means also giving up some degree of control over the experiment's variables. Not only is Chrome one of the more resources-heavy modern browsers regarding CPU and memory utilization [30], but there are cases of unintuitive and unexpected behaviour both on a configuration[1] and runtime level[2]. Therefore, these experiment results should be treated in the context of having unknown variables influencing them.

7.3 Results

In this section, we present the results of our experiments. These findings provide a comprehensive understanding of the performance and behavior of our MASQUE implementation under various conditions.

7.3.1 QUIC Baseline

We first measure the throughput of mvfst on a client ↔ server connection using the QUIC interop runner[3]. This allows us to have an understanding of the limit of potential throughput.

Runs on bitcoingold and dogecoingold via `eno5` yield an average result of 615Mbit/s with a deviation of 39Mbit/s.

7.3.2 TunConnectIP

In the TunConnectIP experiment, we observe a clear trend related to the number of hops and the resulting throughput (see figure 7.4):

- **1 Hop**: The throughput stabilizes between approximately 320 and 400 Mbit/s.
- **2 Hops**: A noticeable decrease, with throughput ranging from 230 to 340 Mbit/s.
- **3 Hops**: Further reduction to between 170 and 280 Mbit/s.
- **4 Hops**: The lowest throughput observed, varying between 110 and 250 Mbit/s.

[1] `https://bugs.chromium.org/p/chromium/issues/detail?id=447590`

[2] `https://bugs.chromium.org/p/chromium/issues/detail?id=1158402`

[3] `https://github.com/quic-interop/quic-interop-runner`

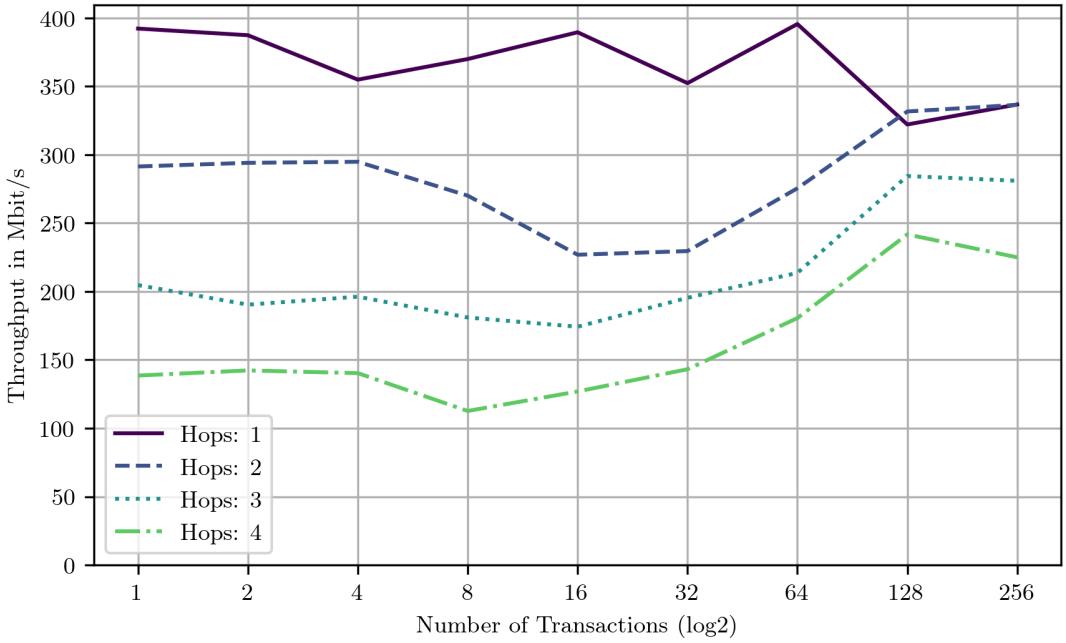

Figure 7.4: TunConnectIP: Number of Transactions vs. Accumulated Throughput (CC: None)

Across all scenarios involving hops, we notice a consistent pattern:

- From 1 to 4 transactions (T=1 to T=4), the throughput remains relatively stable.
- There is a slight decline in performance after T=4, followed by an increase after T=32.
- Beyond T=128, the throughput plateaus, showing no significant changes.

The observed variations in throughput across different hop counts and transaction numbers can be attributed to the library-specific handling of multiple transactions. Notably, in our implementation using mvfst, all transactions share the same QUIC connection, thread, and `epoll` instance. This architectural choice has a direct impact on performance, as evidenced by the CPU usage patterns. Lower throughput often correlates with reduced CPU usage, suggesting periods of idleness likely caused by the scheduling behavior of mvfst.

We clarify these phenomena further in the HTTPConnectIP and HTTPConnectUDP experiments, where the absence of external measurement applications leads to a more controlled environment with fewer unknown variables.

The provided flamegraphs (see figures 7.5 and 7.6) offer a visual representation of exemplary performance results for the MASQUE client and proxy server (hops: 1, T: 64).

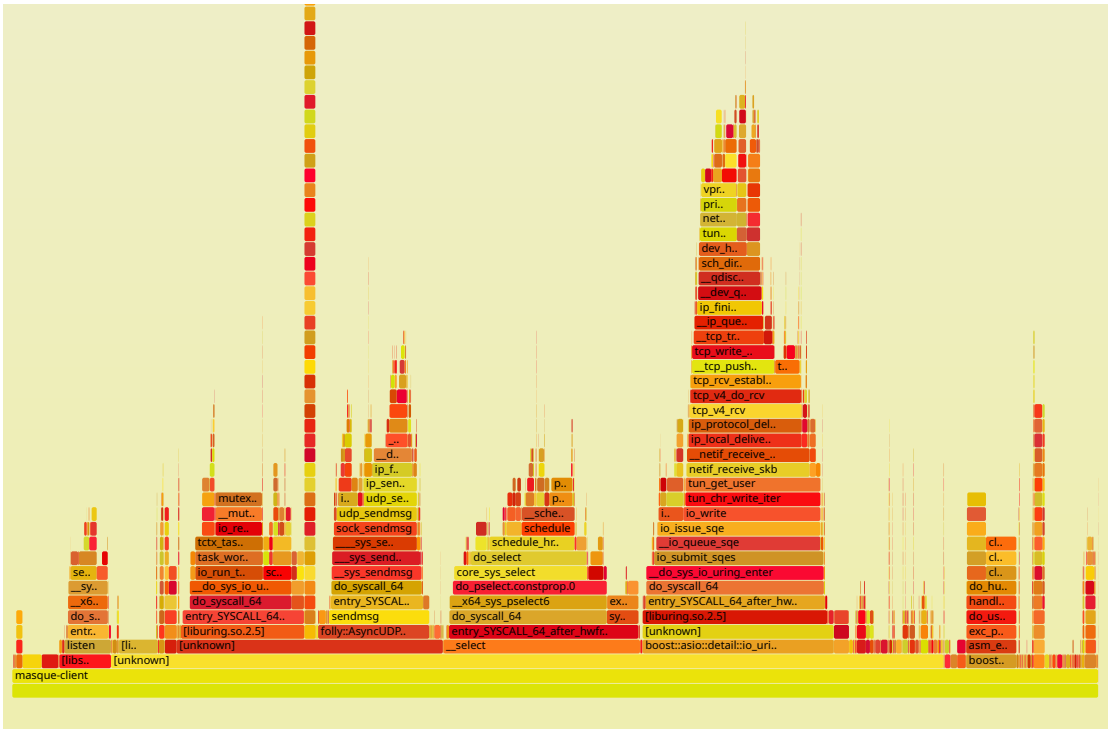

Figure 7.5: TunConnectIP: MASQUE Client Flamegraph (Hops: 1, Transactions: 64, CC: None)

A key observation is the relatively low overhead associated with the client and server TUN devices, a result of the efficient implementation using `io_uring`. This efficiency is crucial for ensuring that the system can handle general-purpose uses without significant performance degradation.

The flamegraphs illustrate the distribution of computational resources across different functions, with a notable concentration in areas related to network processing and data handling. This distribution aligns with the expectations for a system primarily engaged in network data transfer and proxying activities.

In summary, the TunConnectIP experiment demonstrates a clear impact of the number of hops on the overall system throughput, with more hops leading to lower throughput. The observed performance patterns are closely tied to the underlying implementation details of mvfst and the handling of multiple transactions.

7.3.3 HTTPConnectIP / HTTPConnectUDP

We analyze HTTPConnectIP and HTTPConnectUDP, noting that while their values are mostly similar across all metrics, HTTPConnectUDP typically performs slightly better. This advantage is due to its lower packet overhead and the use of UDP sockets,

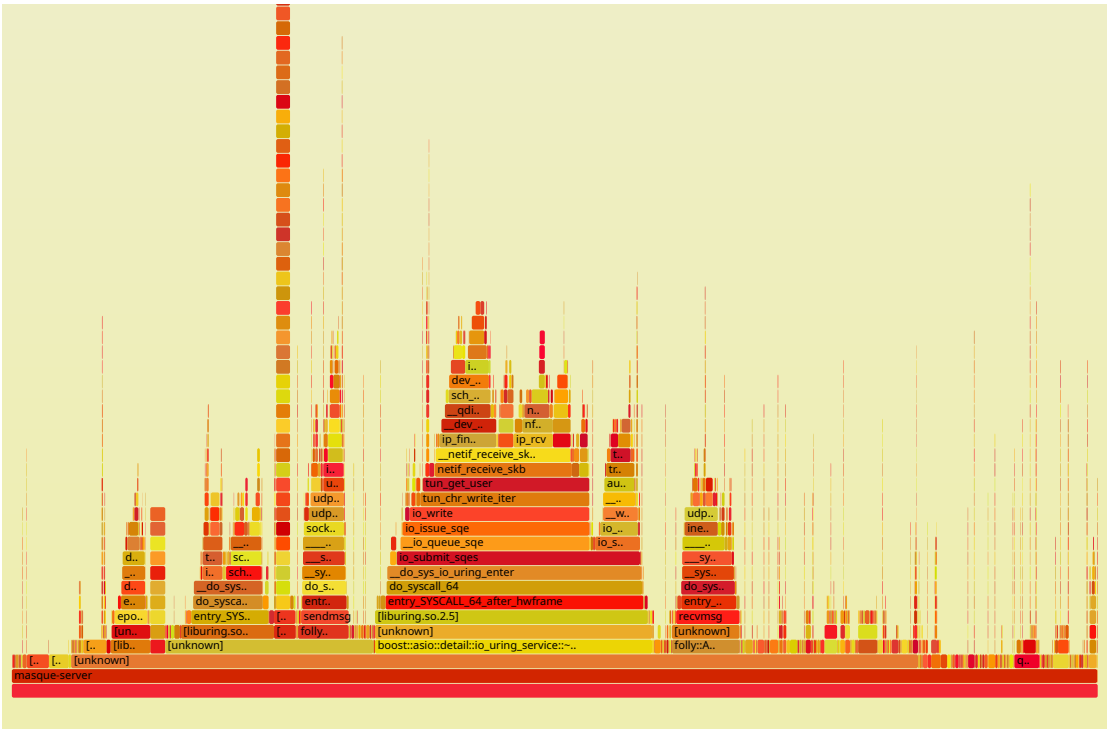

Figure 7.6: TunConnectIP: MASQUE Server Flamegraph (Hops: 1, Transactions: 64, CC: None)

which are less resource-intensive than TUN devices. Consequently, our focus is on HTTPConnectIP for a detailed examination of the results.

Throughput Analysis

Referring to Figure 7.7, we observe:

- Across all hop scenarios, HTTPConnectIP achieves higher throughput than TunConnectIP, expectedly due to the dedicated HTTP client. Specifically, we see:
  - 480 to 830 Mbit/s for 0 hops.
  - 340 to 550 Mbit/s for 1 hop.
  - 240 to 420 Mbit/s for 2 hops.
  - 150 to 340 Mbit/s for 3 hops.
- The significant increase in throughput from T=1 to T=2, especially at 1 hop, is noteworthy. At 0 hops, the limiting factor is the HTTP server's capacity to send data at T=1, as indicated by the CPU usage in Figures A.7 and A.8.

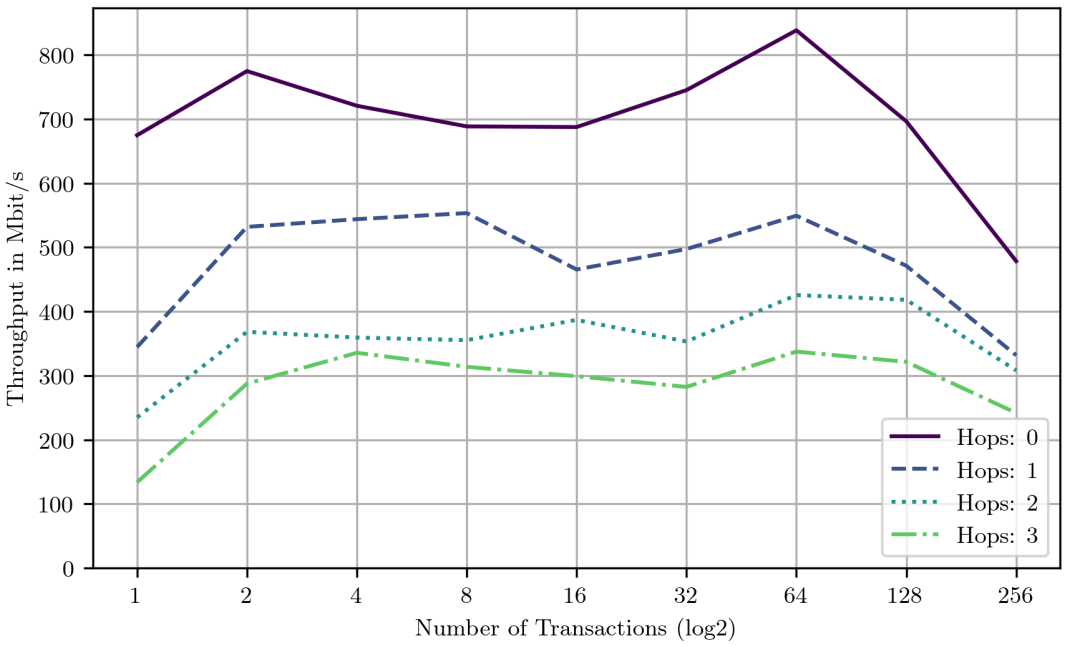

Figure 7.7: HTTPConnectIP: Number of Transactions vs. Acc. Throughput (CC: Cubic)

- The slight decrease in throughput between T=8 and T=32, followed by an increase at T=64 in all hop scenarios, suggests mvfst's internal scheduling behavior. The minimal difference in HTTP server usage at T=64 supports this (see Figures A.20, A.21, and A.22).
- Beyond T=64, we notice a decrease in throughput, indicating the limitations in scalability due to overhead from numerous parallel transactions.

TTFB and RTT

- In Figure 7.8, QUIC-level TTFB remains consistent across transaction numbers, with a gradual increase per additional hop.
- RTT, as shown in Figure 7.9, shows an upward trend in RTT from T=1 to T=256 for all hops, with the most stable increase happening between T=16 and T=128. Fluctuations at T=64, especially at 0 hops, suggest underlying library dynamics.

Latency and Jitter

- Latency trends in Figure 7.10 mirror those of RTT. Given the duplex cable setup and GET requests' download-centric nature, this alignment is expected.

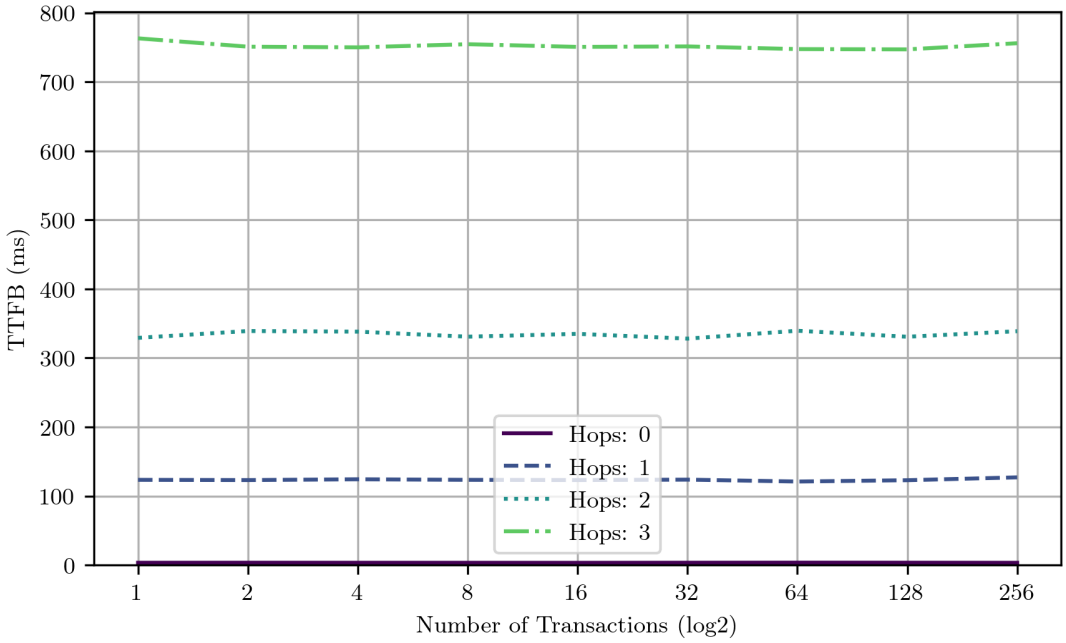

Figure 7.8: HTTPConnectIP: Number of Transactions vs. QUIC TTFB (CC: Cubic)

- Jitter analysis in Figure 7.11 shows initial stability up to T=16, followed by a rise at T=256. For 0 hops, jitter starts at approximately 0.025ms, increasing to 0.06ms at T=256. For 3 hops, it begins higher and exhibits a similar upward trend. This behavior is aligned with the CPU usage patterns observed. For example, at 2 hops and T=1, the CPU usage of the client and proxies indicates different levels of packet handling and buffering, impacting latency and jitter (see Figures Figures A.23 to A.26).

Individual HTTP TTFB

- The individual HTTP TTFB for different transactions, shown in Figure 7.12, demonstrates minimal variance across hops. The TTFB per transaction increases with the number of transactions, and this increase is proportional. We also observe a significant rise in variance, indicating that some transactions receive their first byte as quickly as in runs with fewer transactions.

HTTPConnectUDP Comparison

HTTPConnectUDP presents similar outcomes to HTTPConnectIP in all metrics, including CPU usage patterns (see Chapter A for detailed CPU logs). Figures Figures A.1 to A.6 detail these observations.

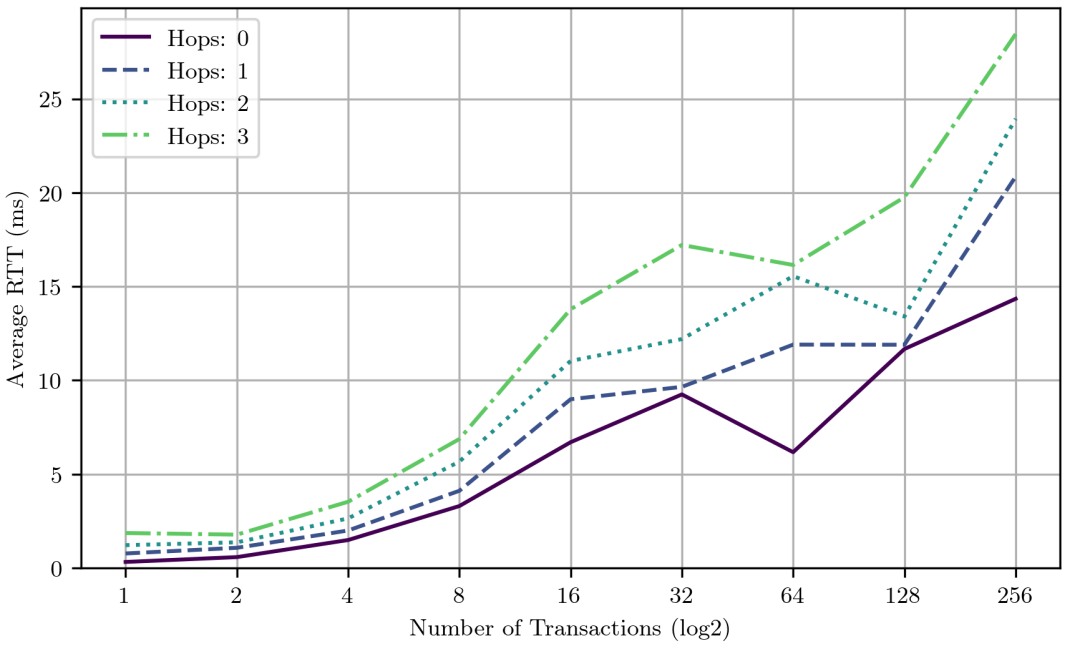

Figure 7.9: HTTPConnectIP: Number of Transactions vs. RTT (CC: Cubic)

In summary, our experiments with HTTPConnectIP and HTTPConnectUDP clearly demonstrate that the number of hops in a network connection significantly affects throughput, with more hops leading to reduced throughput. Additionally, the handling of multiple transactions has a notable impact on TTFB, RTT, latency, and jitter.

7.3.4 SeleniumConnectIP

Despite encountering outliers, likely influenced by Chrome's unpredictable behavior, the SeleniumConnectIP results demonstrated clear trends, particularly regarding the impact of transaction and client numbers on page loading time and TTFB.

Crucially, the analysis showed that CPU bottlenecking was not a primary factor, as detailed in the CPU usage studies for various transaction scenarios (T: 2, 4, 8). The referenced figures (Figures A.37 to A.42) consistently showed that no experiment surpassed 70% CPU usage on the client side, underlining that these scenarios did not yet involve a sufficient number of Selenium instances to cause a CPU bottleneck. Nevertheless, outliers were still observed, suggesting the influence of factors other than CPU performance constraints in these cases.

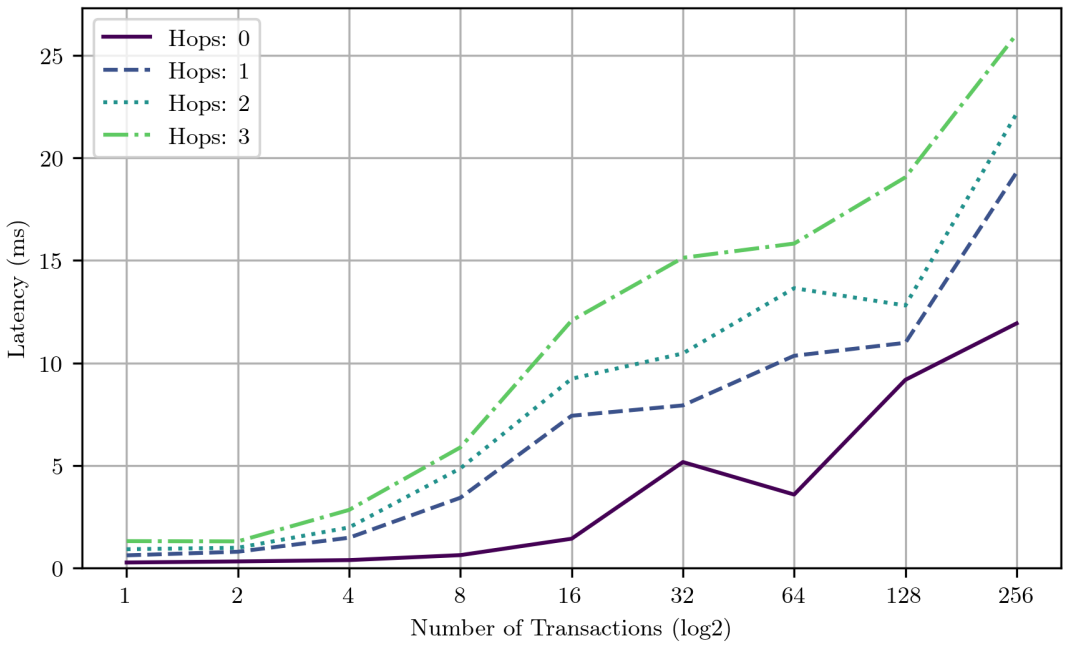

Figure 7.10: HTTPConnectIP: Number of Transactions vs. Latency (CC: Cubic)

Page Loading Time Analysis

Analyzing page loading times under different network conditions, as illustrated in 7.13, 7.14, and 7.15, reveals several key points:

- Irrespective of the hop count, there's a marked escalation in both the median page load time and the variance of load times with an increase in transactions and clients. This trend confirms a direct link between heightened network activity and decreased page loading performance.
- More hops lead to a faster increase in load times and their variance. Conversely, in scenarios with fewer hops, a higher number of clients and transactions is required to significantly affect median page load times.

TTFB on HTTP Level

Investigating TTFB at the HTTP level, as shown in Figures Figures 7.16 to 7.18, yields different insights:

- In contrast to the page loading time patterns, TTFB demonstrates remarkable stability across varying transaction and client numbers. From T=1 to T=16, the median TTFB experiences only a slight increase, regardless of the hop count.

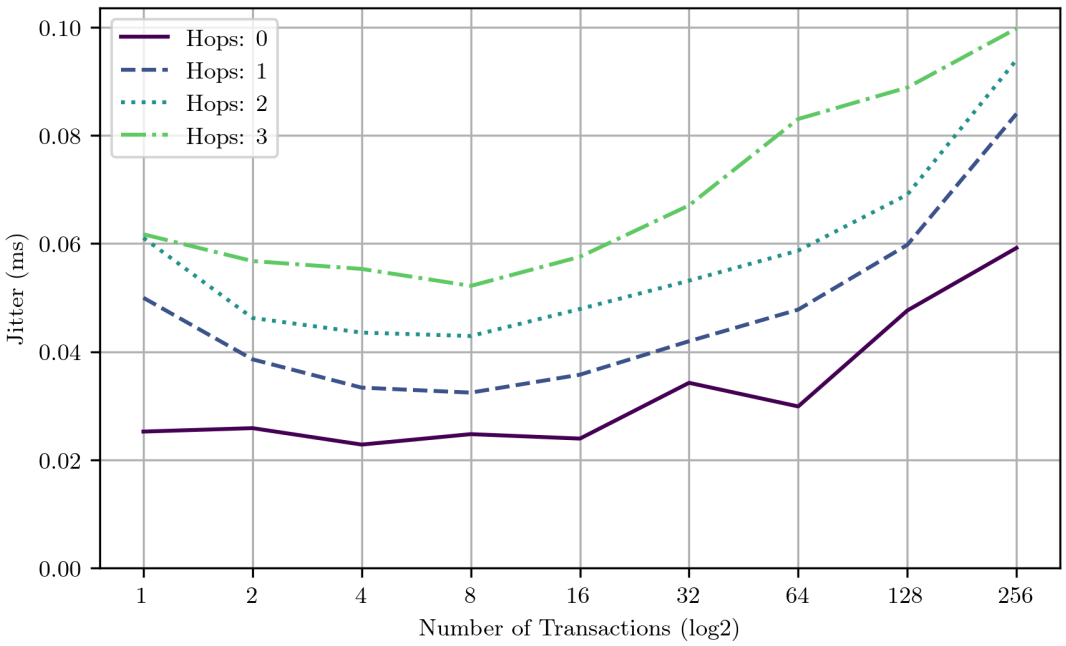

Figure 7.11: HTTPConnectIP: Number of Transactions vs. Jitter (CC: Cubic)

- Although there is an uptick in variance with more transactions and clients, it is less dramatic compared to the page loading times.

This observation indicates that in the context of our MASQUE proxying setup, the overall process of page loading is significantly influenced by the number of proxy hops and the complexity of the proxying mechanism. However, the initial stages of HTTP requests, as denoted by the TTFB, appear less affected. Despite the additional hops and the complexities introduced by the MASQUE proxying, the TTFB remains relatively stable across various test scenarios.

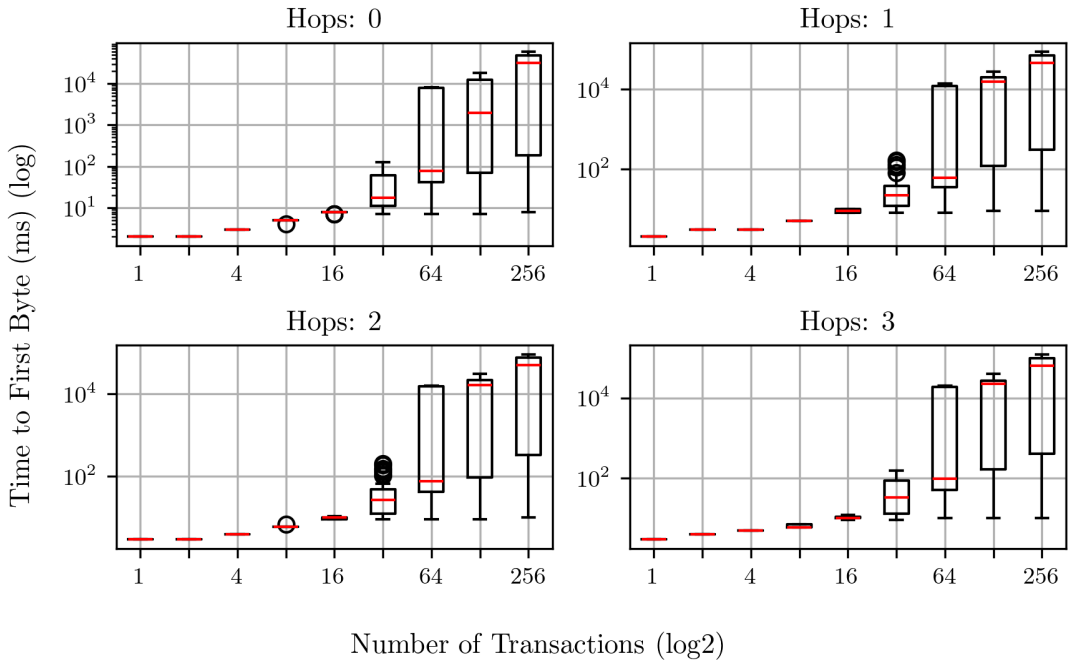

Figure 7.12: HTTPConnectIP: Number of Transactions vs. HTTP TTFB (CC: Cubic)

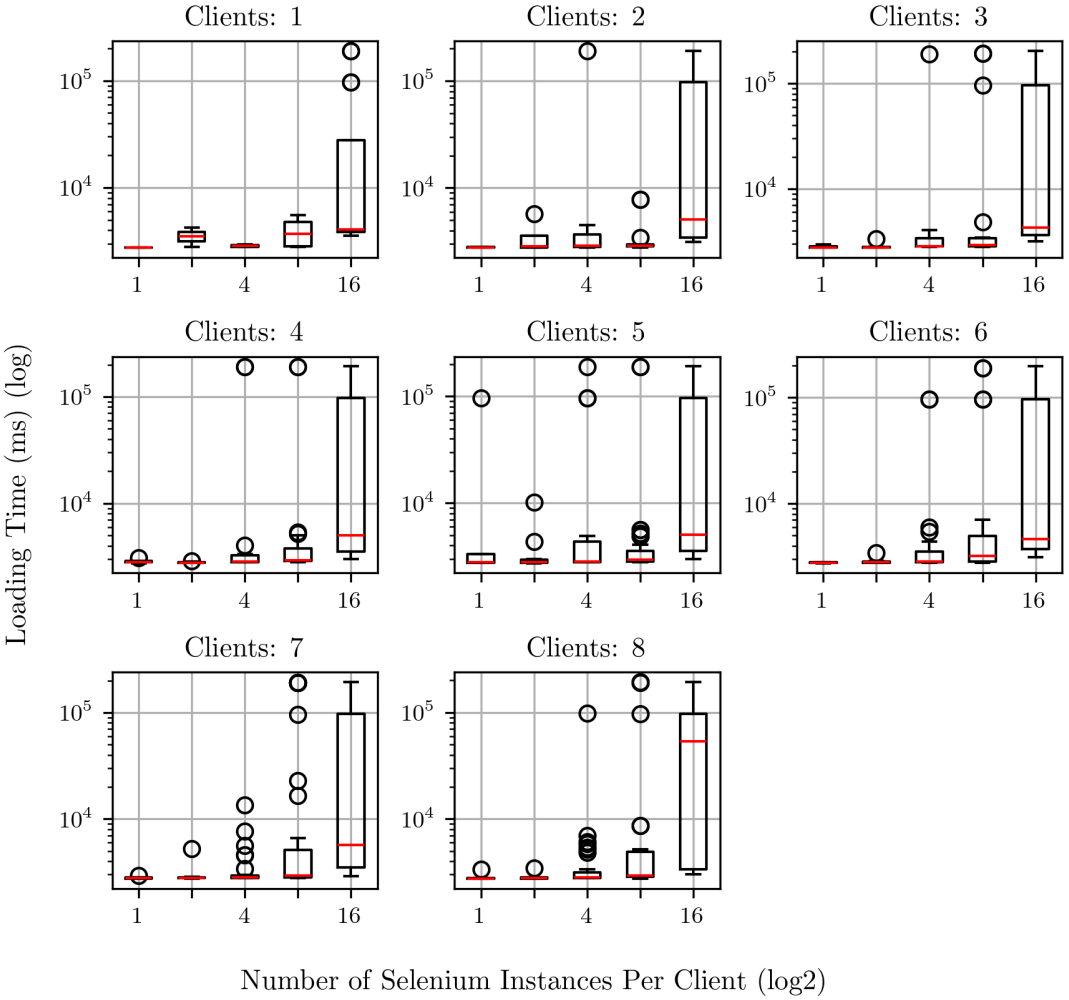

Figure 7.13: SeleniumConnectIP: Number of Transactions / Selenium Instances Per Client vs. Page Loading Time (Hops: 0, CC: Cubic)

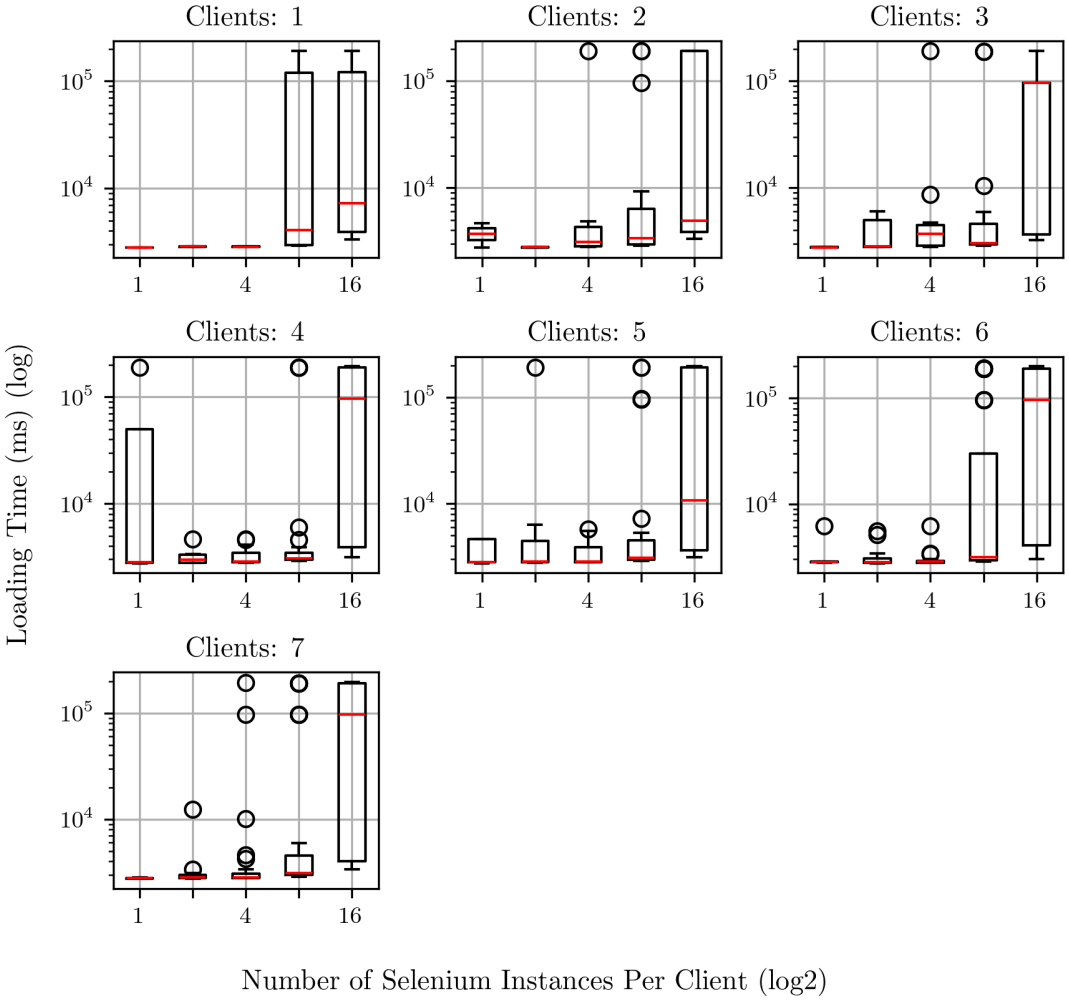

Figure 7.14: SeleniumConnectIP: Number of Transactions / Selenium Instances Per Client vs. Page Loading Time (Hops: 1, CC: Cubic)

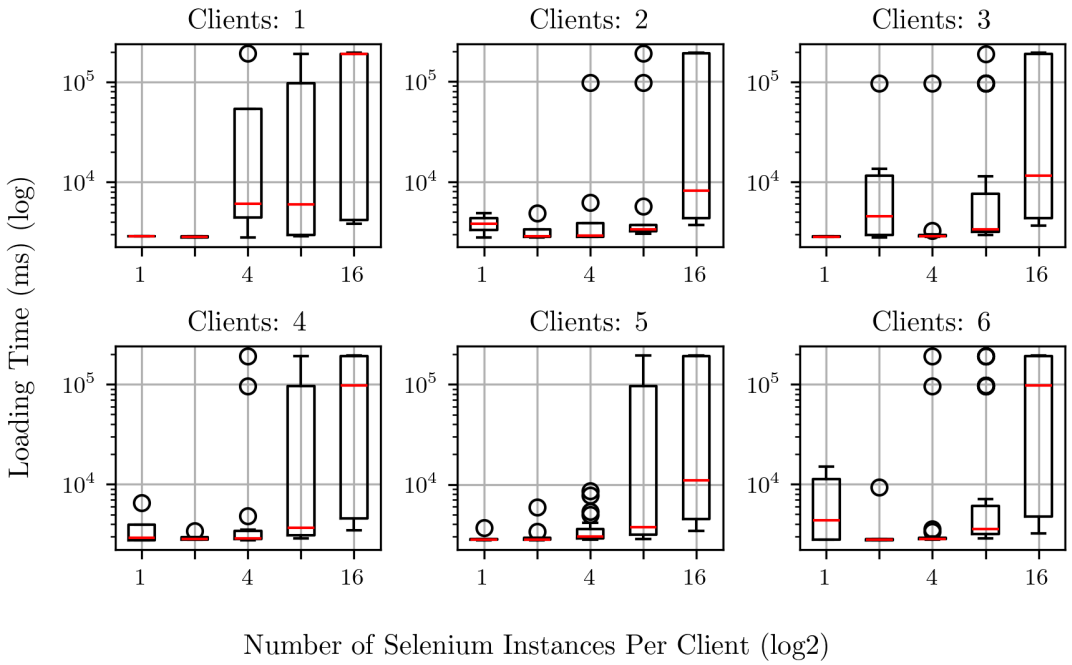

Figure 7.15: SeleniumConnectIP: Number of Transactions / Selenium Instances Per Client vs. Page Loading Time (Hops: 2, CC: Cubic)

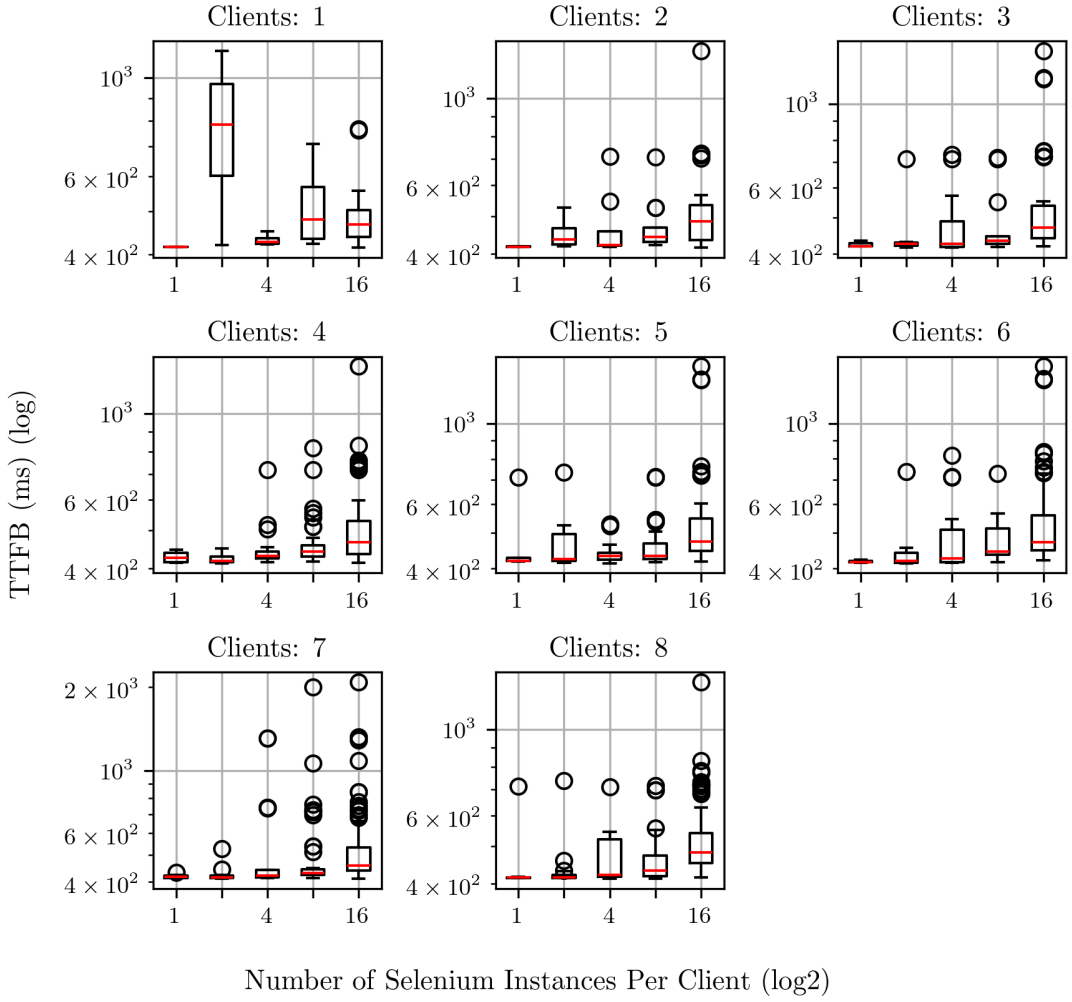

Figure 7.16: SeleniumConnectIP: Number of Transactions / Selenium Instances Per Client vs. HTTP TTFB (Hops: 0, CC: Cubic)

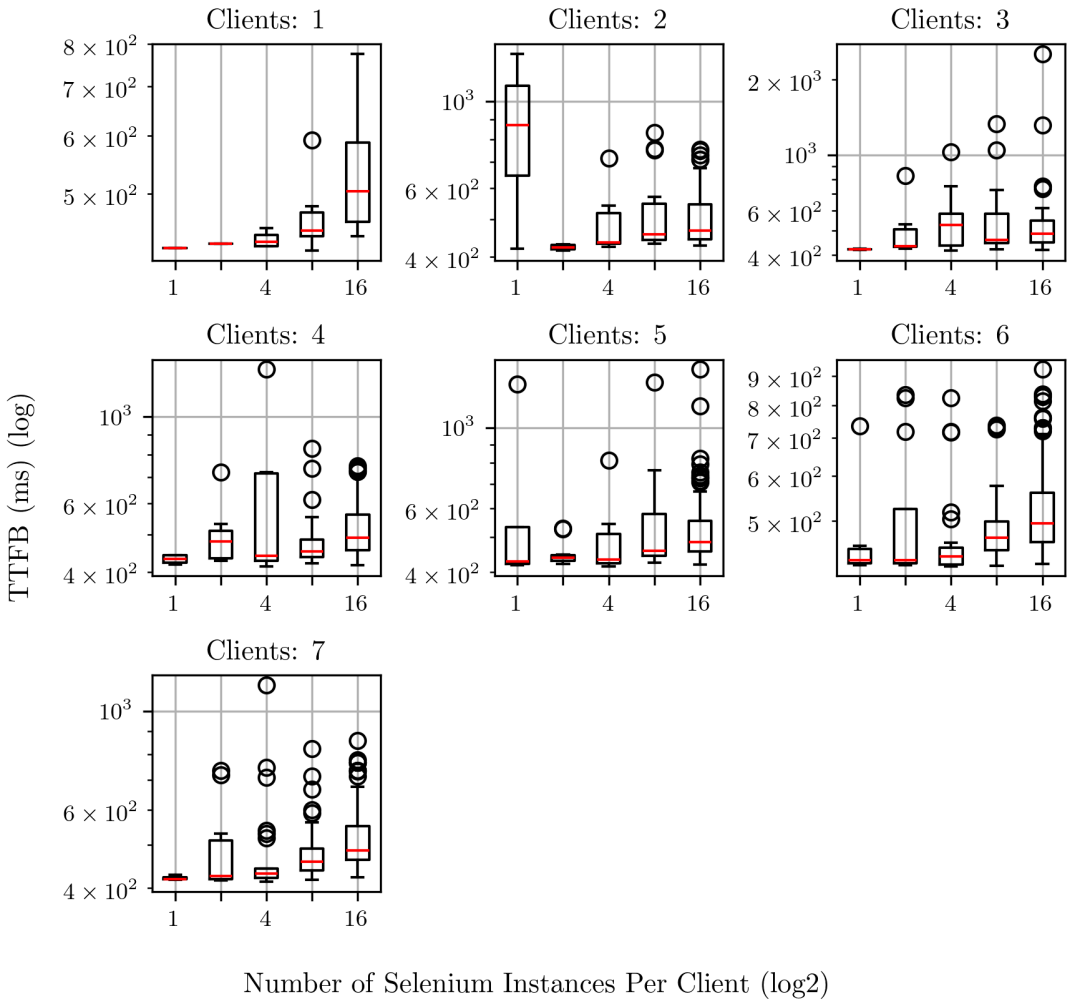

Figure 7.17: SeleniumConnectIP: Number of Transactions / Selenium Instances Per Client vs. HTTP TTFB (Hops: 1, CC: Cubic)

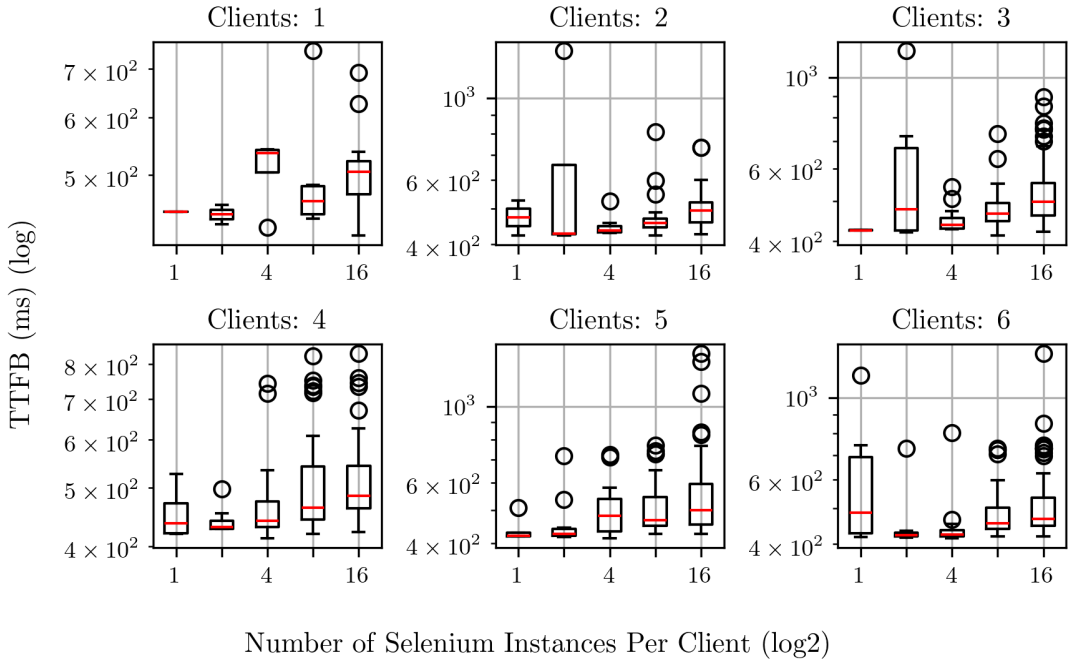

Figure 7.18: SeleniumConnectIP: Number of Transactions / Selenium Instances Per Client vs. HTTP TTFB (Hops: 2, CC: Cubic)

Chapter 8

Conclusion

This project presented the development and evaluation of a custom MASQUE proxy implementation using the mvfst and proxygen libraries. The goal was to analyze MASQUE's capabilities for efficient proxying of IP and UDP traffic.

The experiments demonstrated clear impacts on performance metrics when using multi-hop MASQUE proxying. Each additional proxy hop resulted in reduced throughput, increased TTFB, higher RTT, greater latency, and heightened jitter. These trends were consistent across diverse scenarios.

From an end-user perspective, page load times showed high sensitivity to hop counts and traffic levels during MASQUE proxying. However, the initial TTFB was stable despite the proxying complexity. This suggests that the early request stages are less affected compared to overall page loading.

The implementation handled tunneling and multi-transaction concurrency without excessive resource usage. Optimizations like `io_uring` proved beneficial for the TUN overhead. The modular architecture enabled flexible configurations for diverse test scenarios.

Overall, this project provided empirical insights into real-world MASQUE performance using mvfst and proxygen. It highlighted definite trade-offs between proxy hops and efficiency. The findings can inform future refinements for optimizing and benchmarking MASQUE implementations.

Directions for future work include exploring alternative libraries like new versions of MsQuic[1], improving testbed realism, and comparing MASQUE with other proxy protocols. Focus areas include reducing encapsulation overhead, streamlining multi-hop packet handling, and enhancing concurrency.

Advancing MASQUE efficiency can expand its applicability for security, privacy and circumvention use cases. This project contributed empirical data and practical experience to guide these ongoing development.

[1] Recent analysis has confirmed MsQuic as one of the fastest and most efficient QUIC libraries available, showing significant throughput and CPU utilization improvements compared to alternatives [31].

Chapter A

Appendix

A.1 HTTPConnectUDP Results

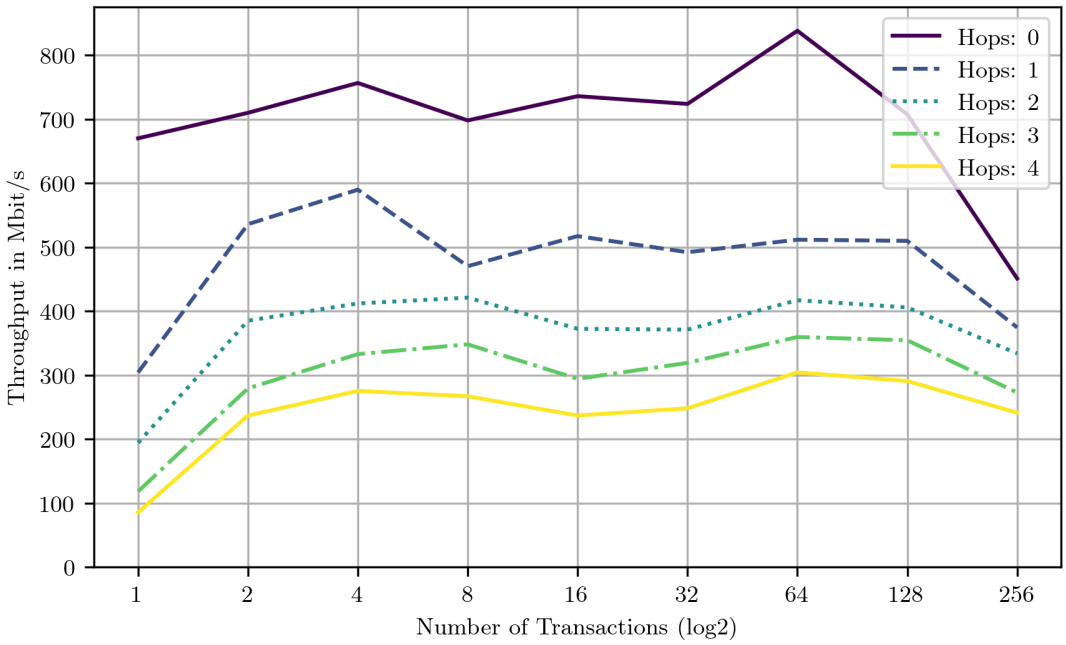

Figure A.1: HTTPConnectUDP: Number of Transactions vs. Acc. Throughput (CC: Cubic)

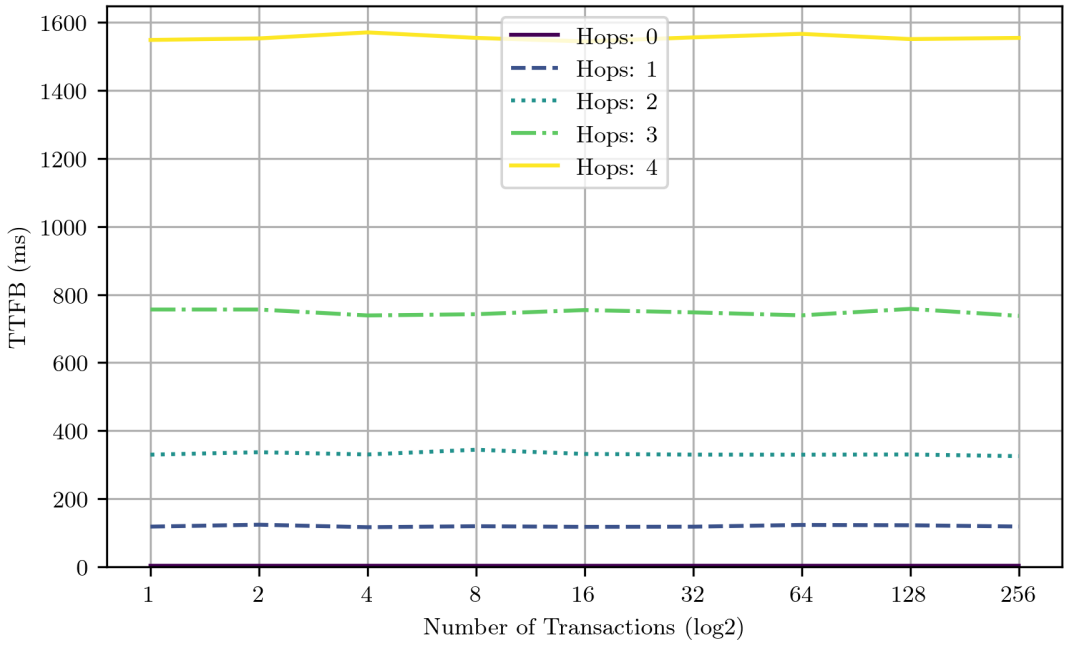

Figure A.2: HTTPConnectUDP: Number of Transactions vs. QUIC TTFB (CC: Cubic)

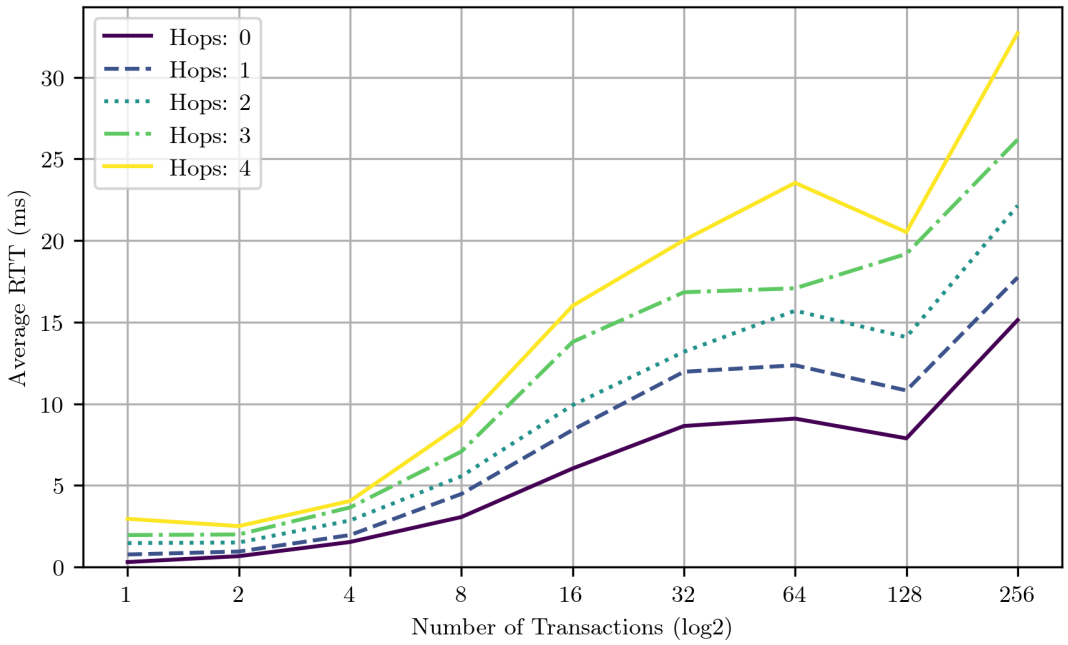

Figure A.3: HTTPConnectUDP: Number of Transactions vs. RTT (CC: Cubic)

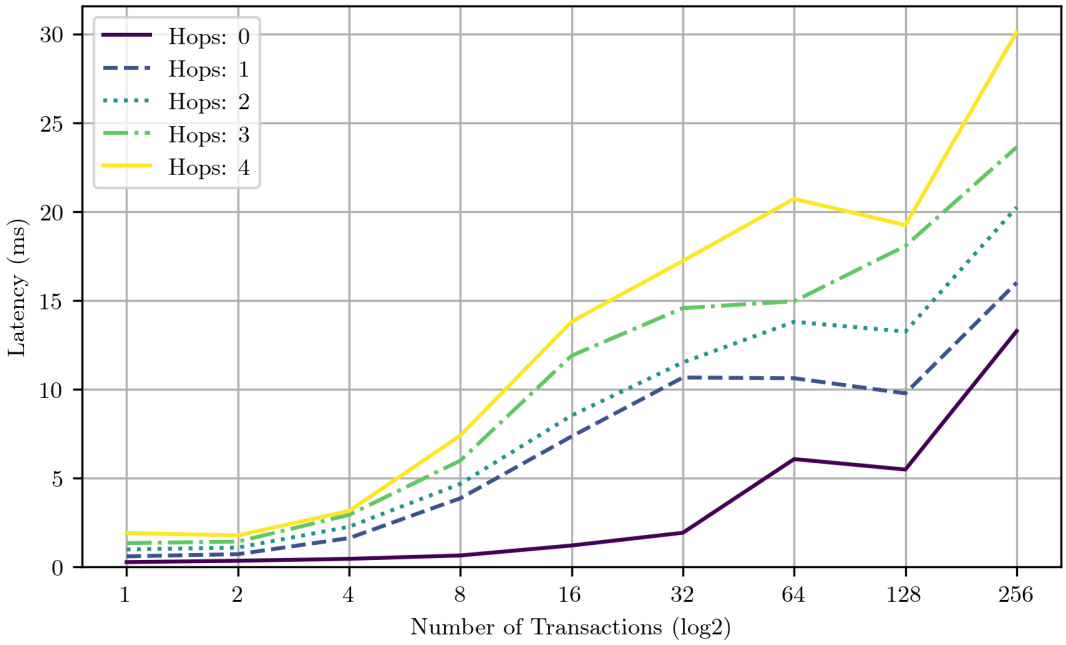

Figure A.4: HTTPConnectUDP: Number of Transactions vs. Latency (CC: Cubic)

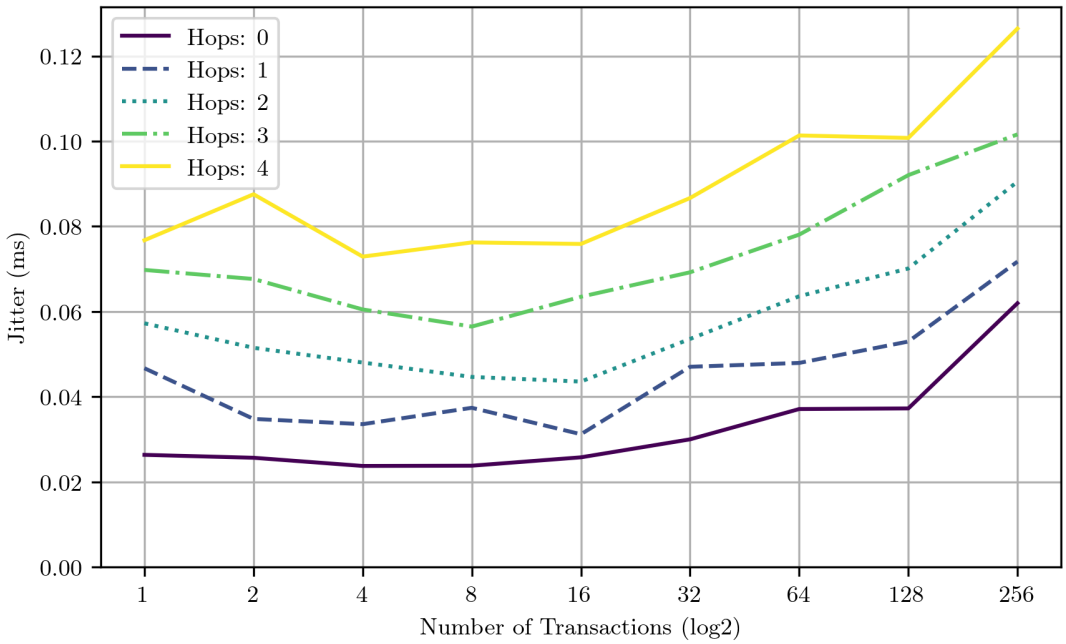

Figure A.5: HTTPConnectUDP: Number of Transactions vs. Jitter (CC: Cubic)

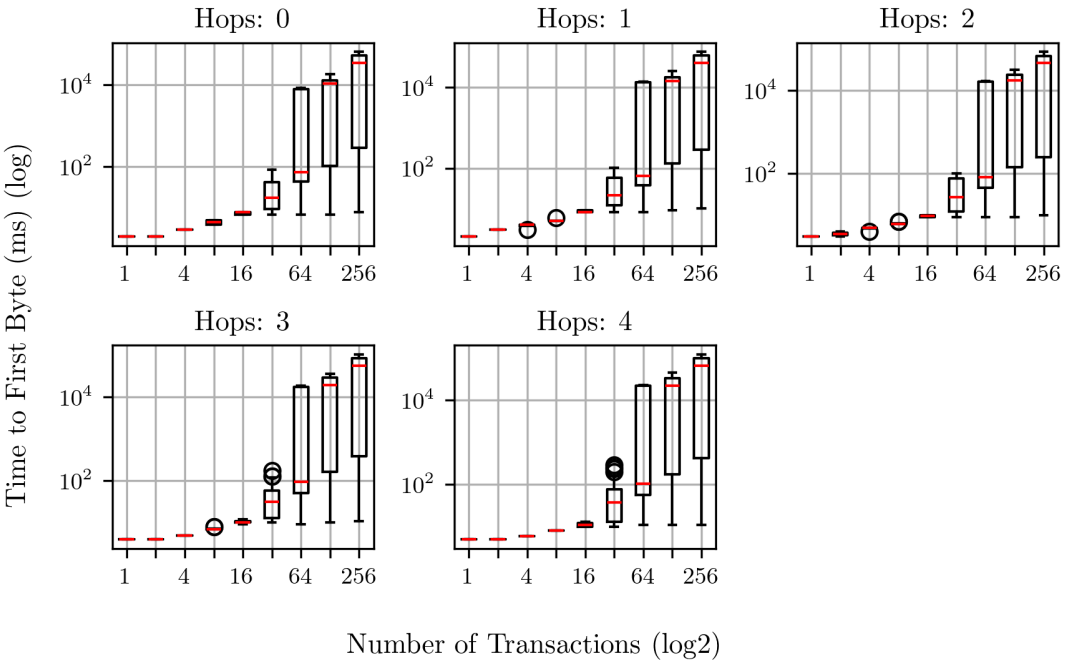

Figure A.6: HTTPConnectUDP: Number of Transactions vs. HTTP TTFB (CC: Cubic)

A.2 Selected CPU Usage for HTTPConnectIP

A.2.1 Hops: 0

T: 1

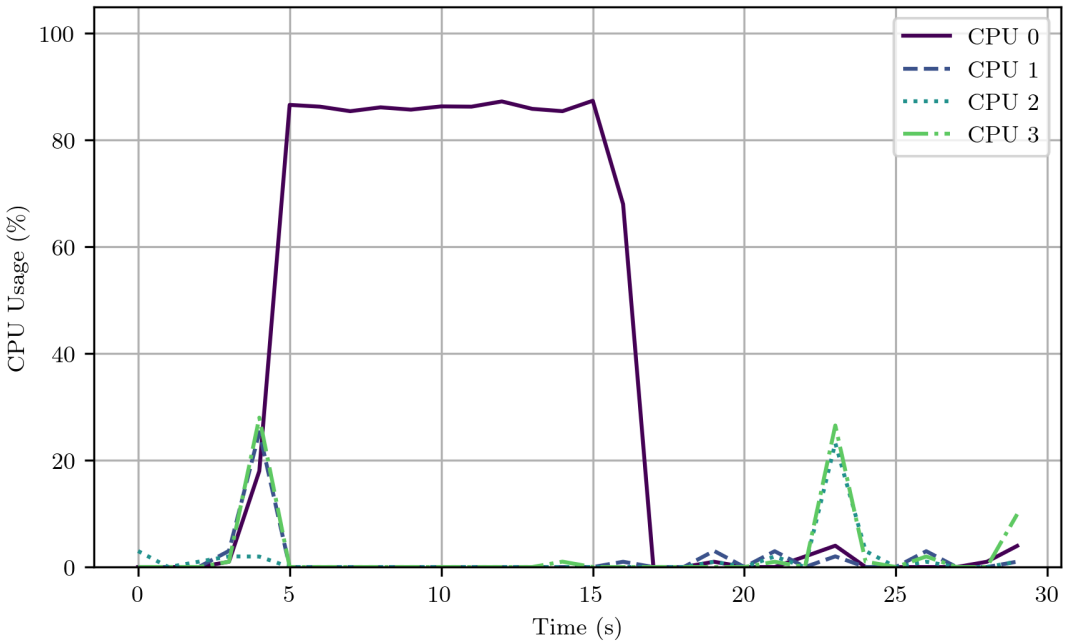

Figure A.7: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 0, T: 1)

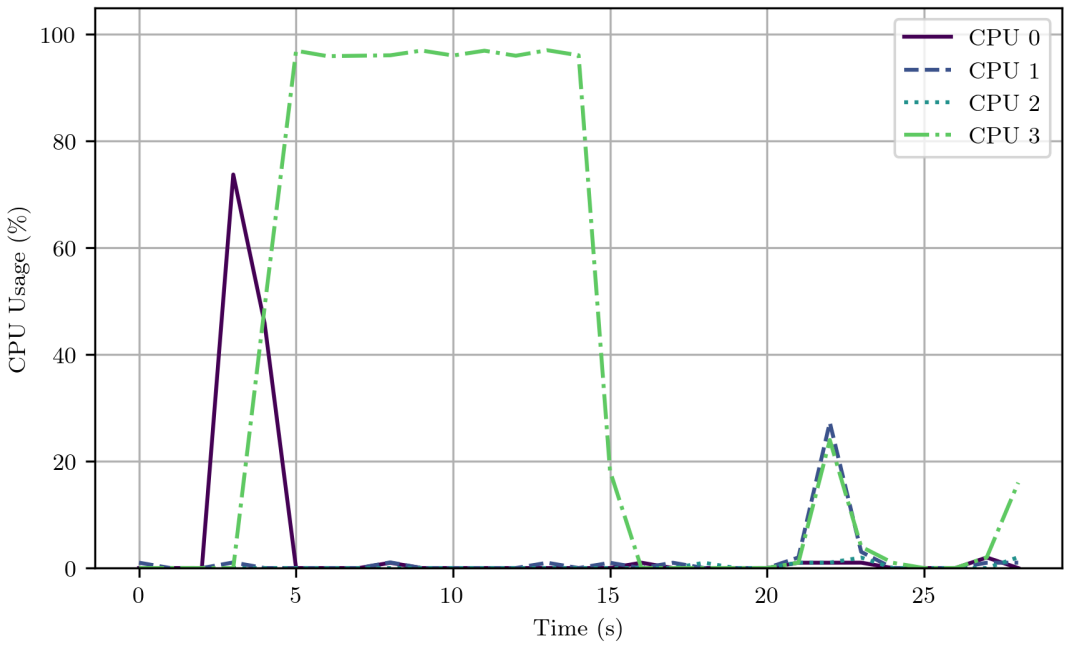

Figure A.8: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 0, T: 1)

T: 2

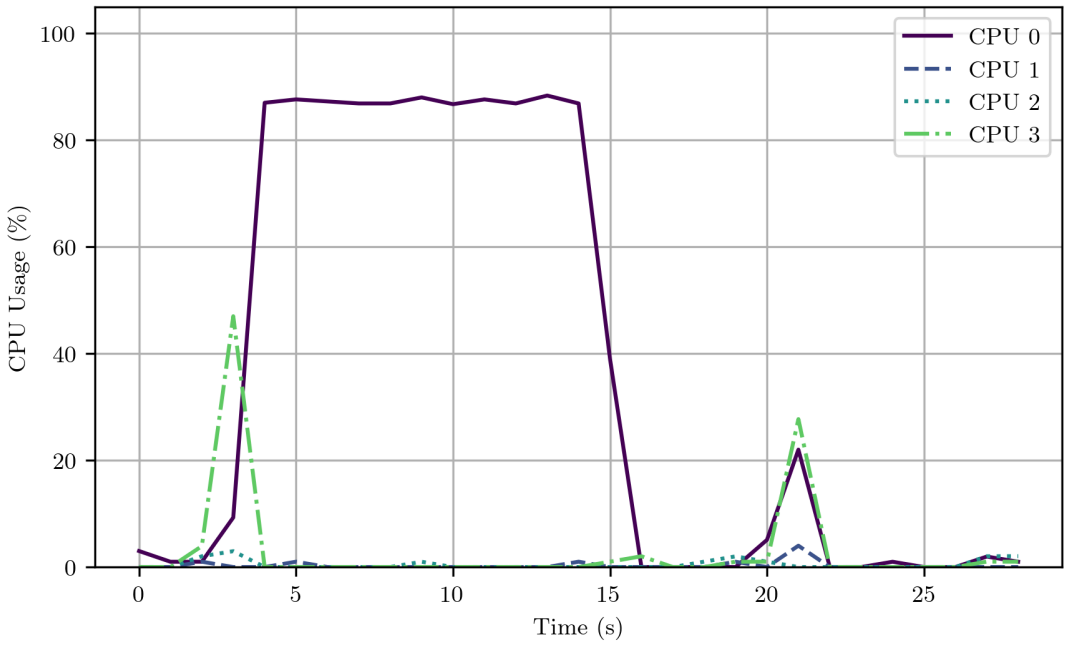

Figure A.9: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 0, T: 2)

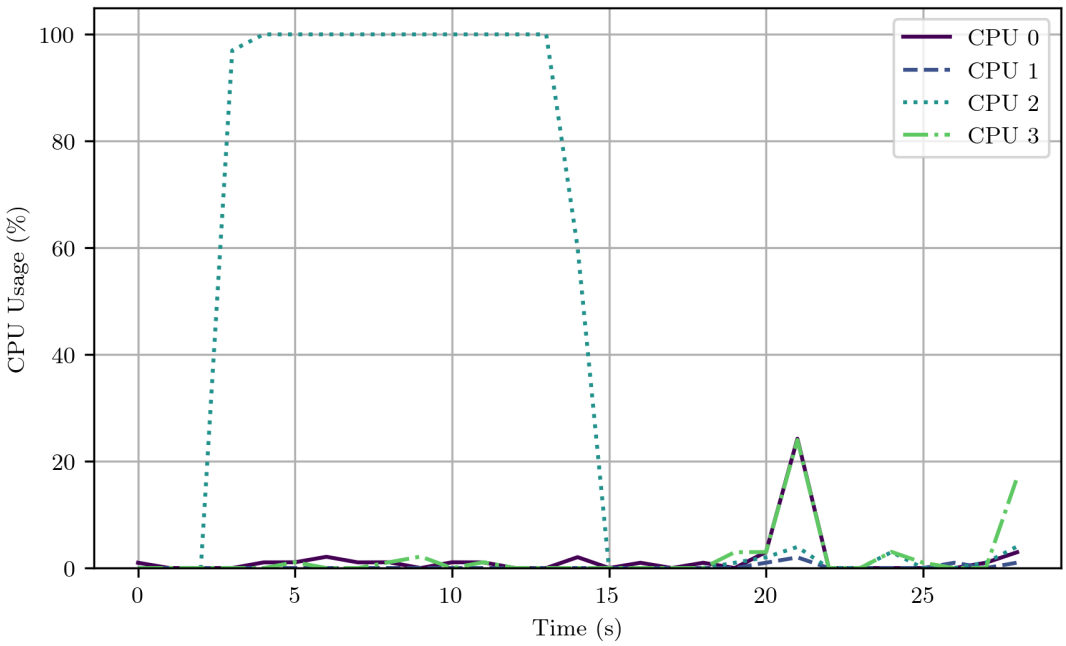

Figure A.10: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 0, T: 2)

A.2.2 Hops: 1

T: 1

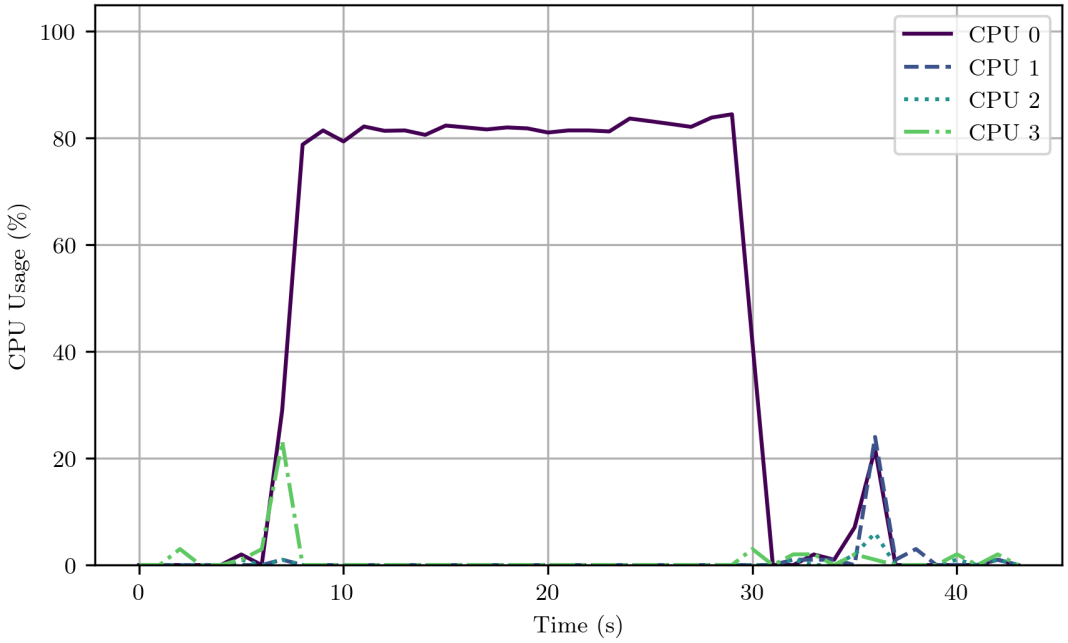

Figure A.11: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 1, T: 1)

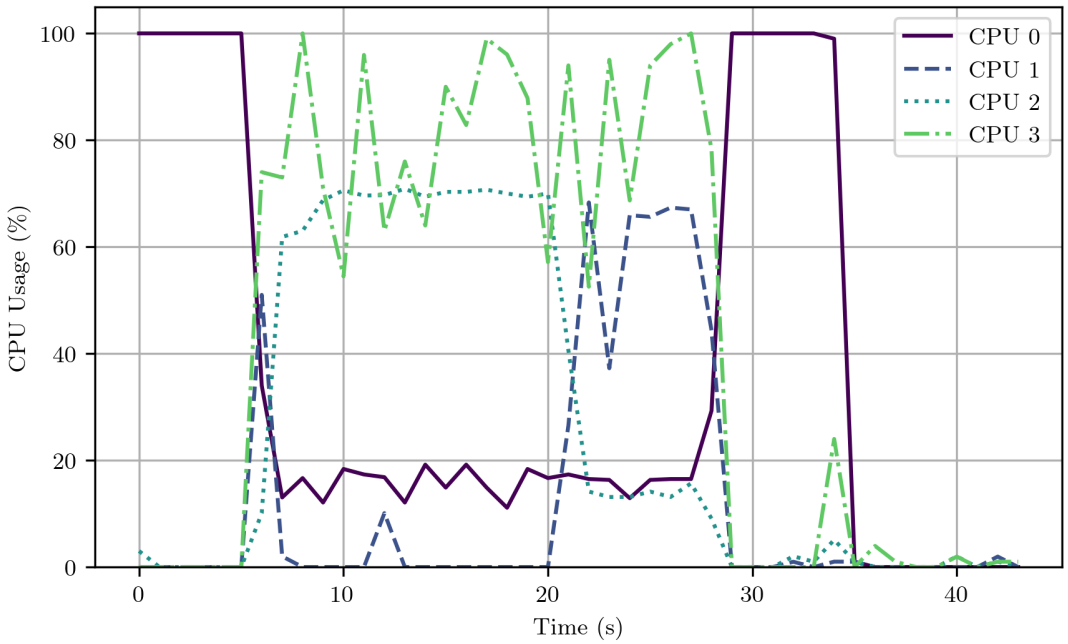

Figure A.12: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Proxy 1, Hops: 1, T: 1)

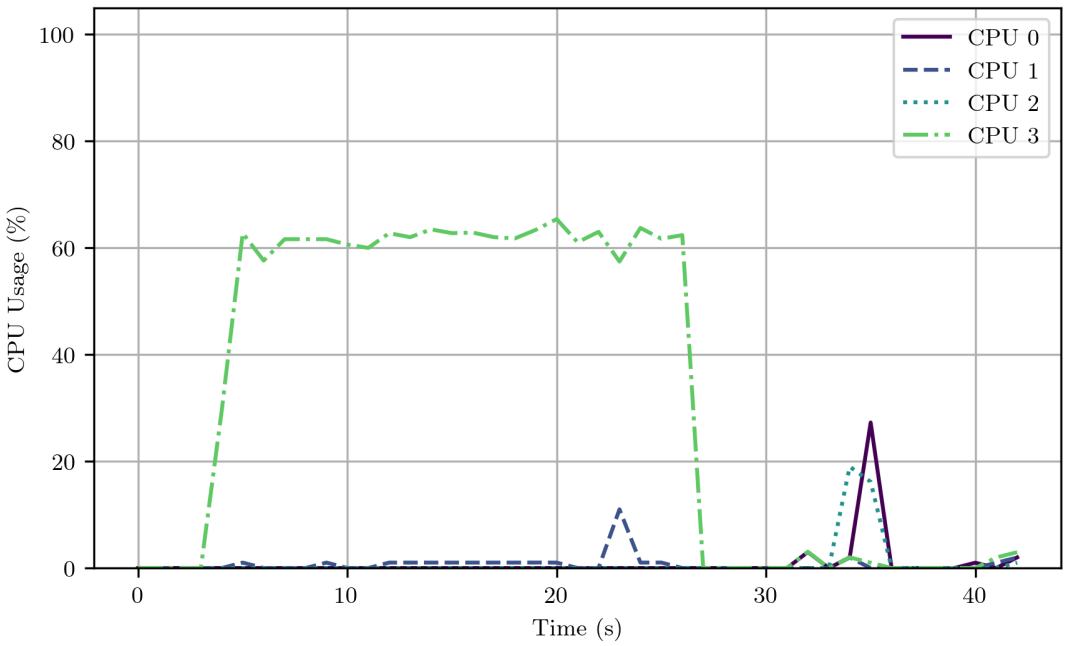

Figure A.13: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 1, T: 1)

T: 2

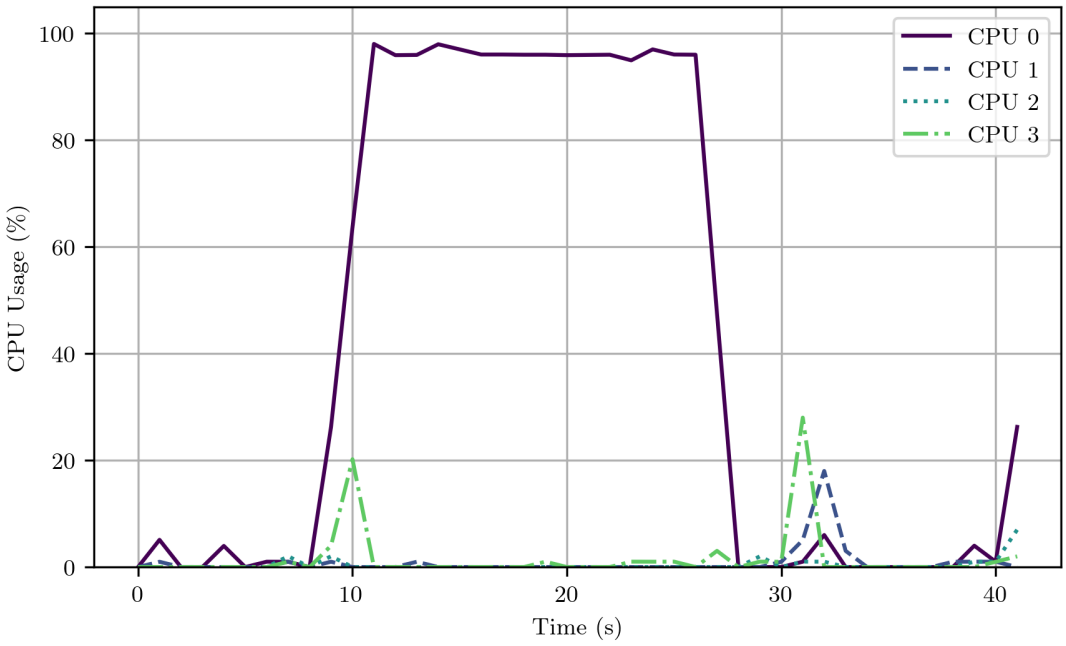

Figure A.14: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 1, T: 2)

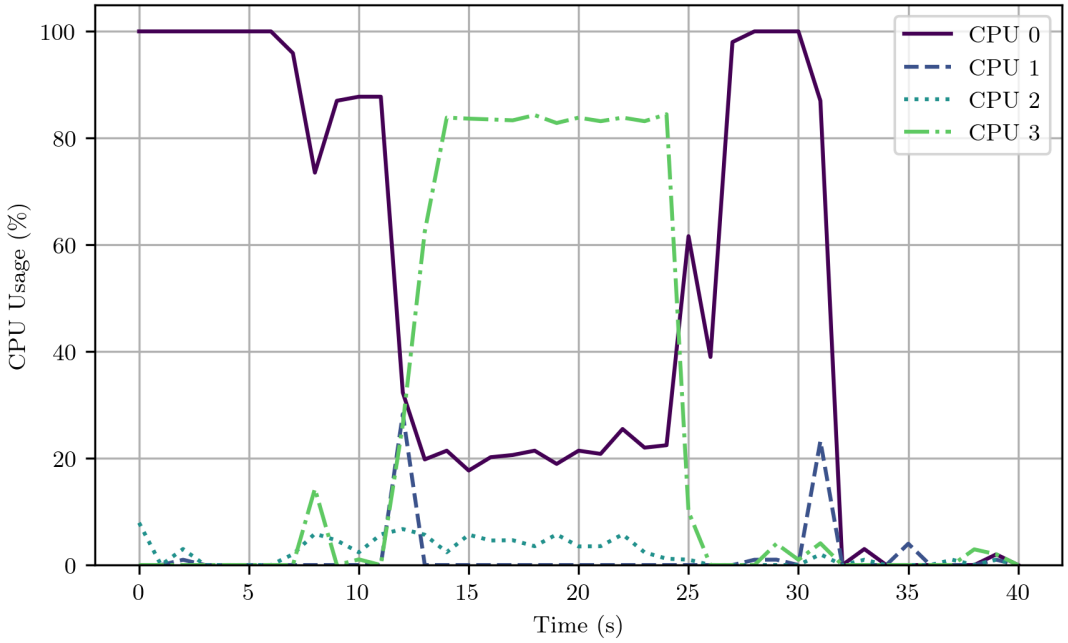

Figure A.15: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Proxy 1, Hops: 1, T: 2)

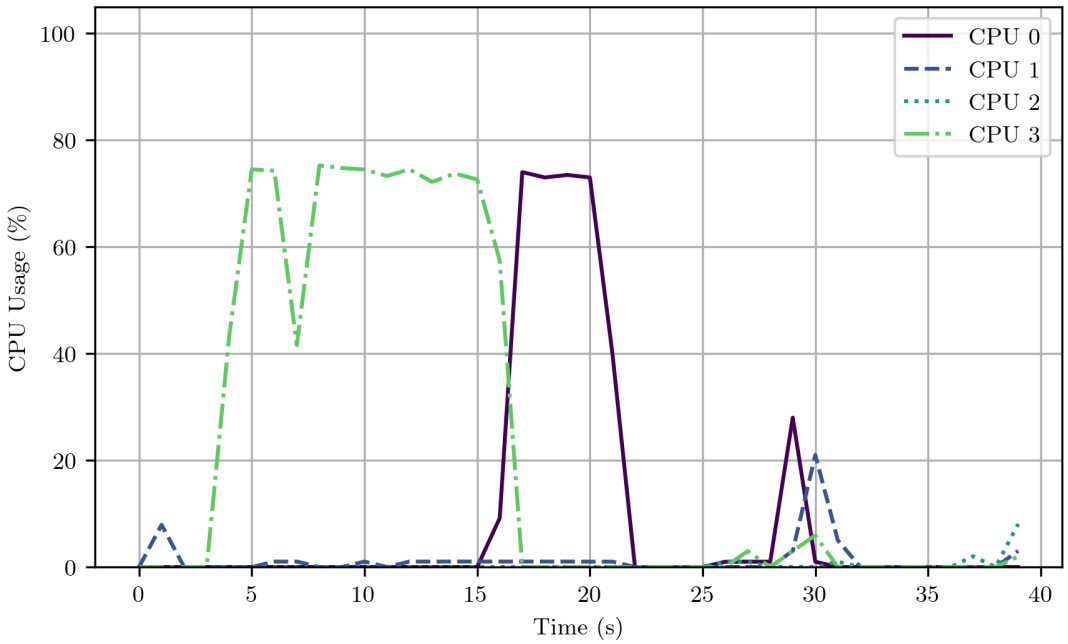

Figure A.16: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 1, T: 2)

T: 16

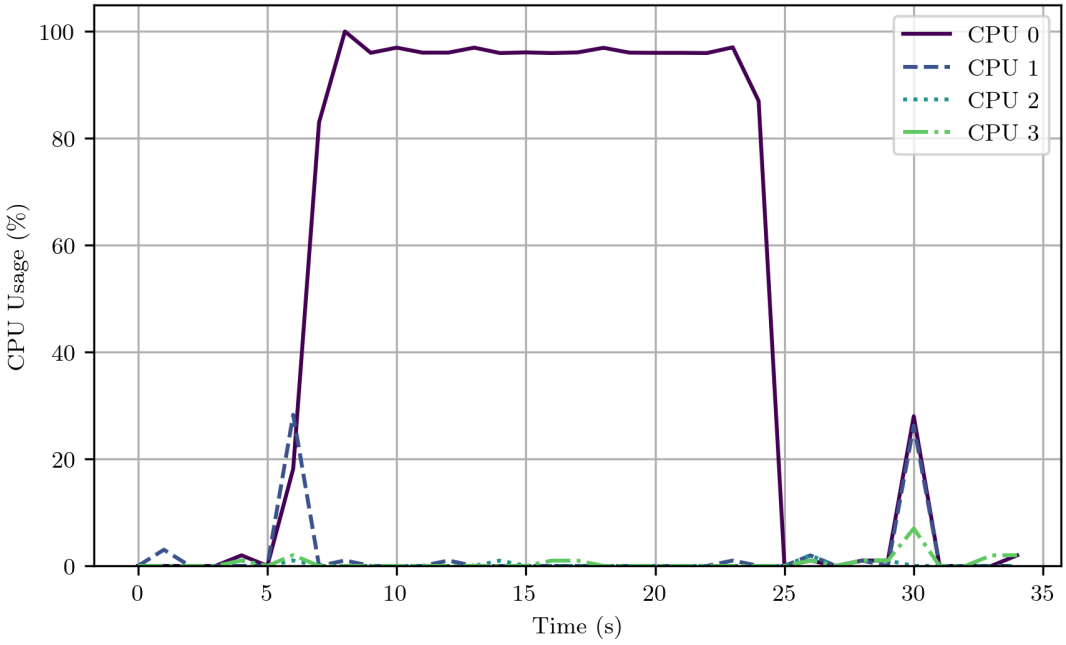

Figure A.17: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 1, T: 16)

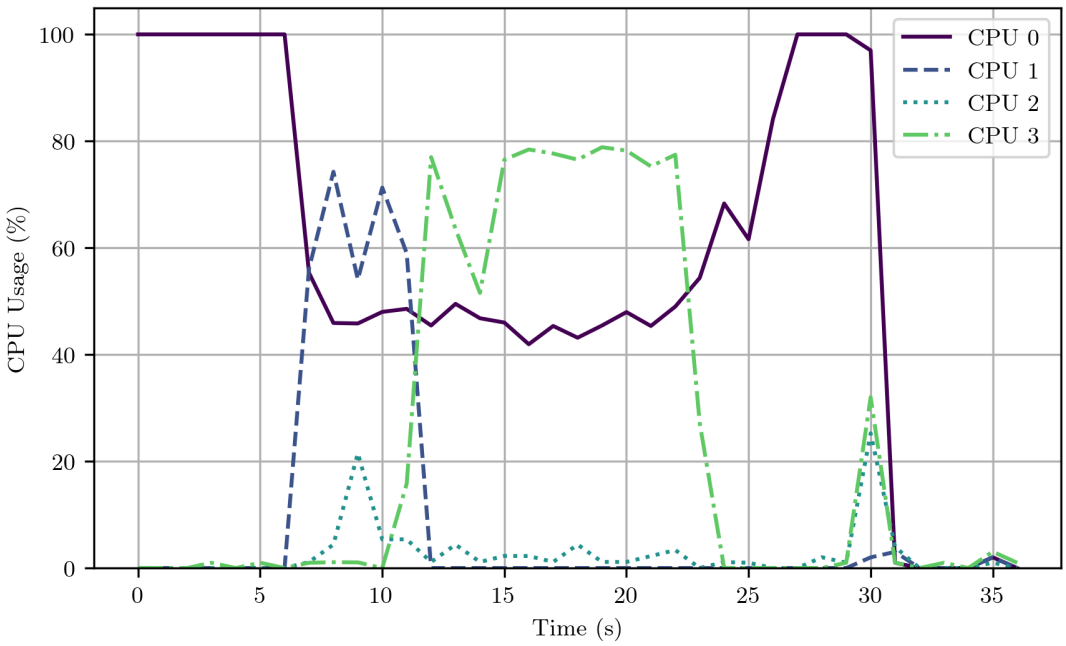

Figure A.18: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Proxy 1, Hops: 1, T: 16)

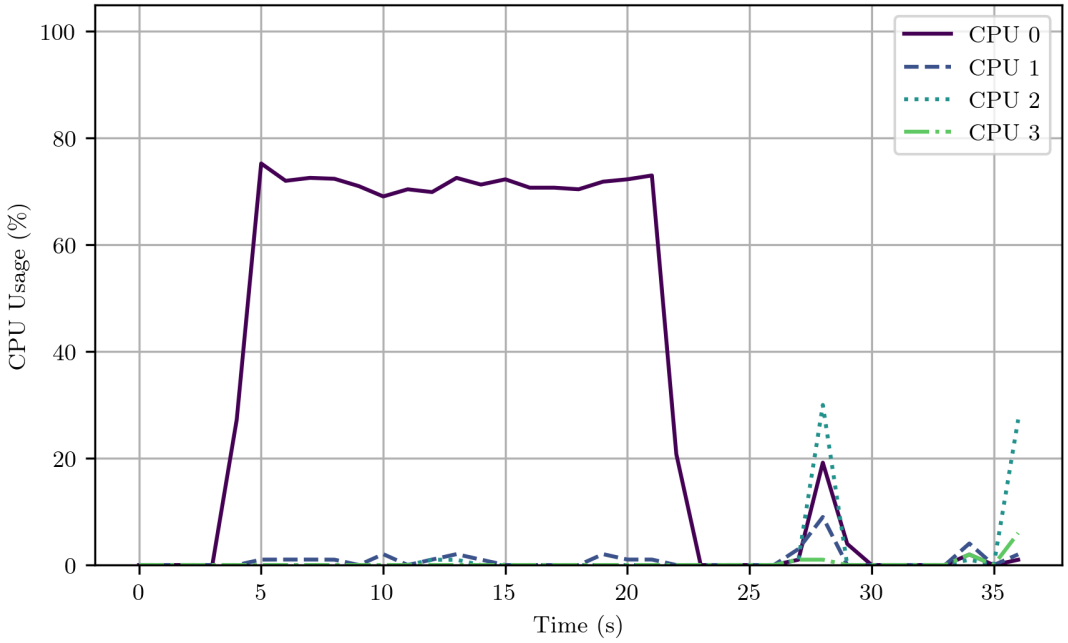

Figure A.19: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 1, T: 16)

T: 64

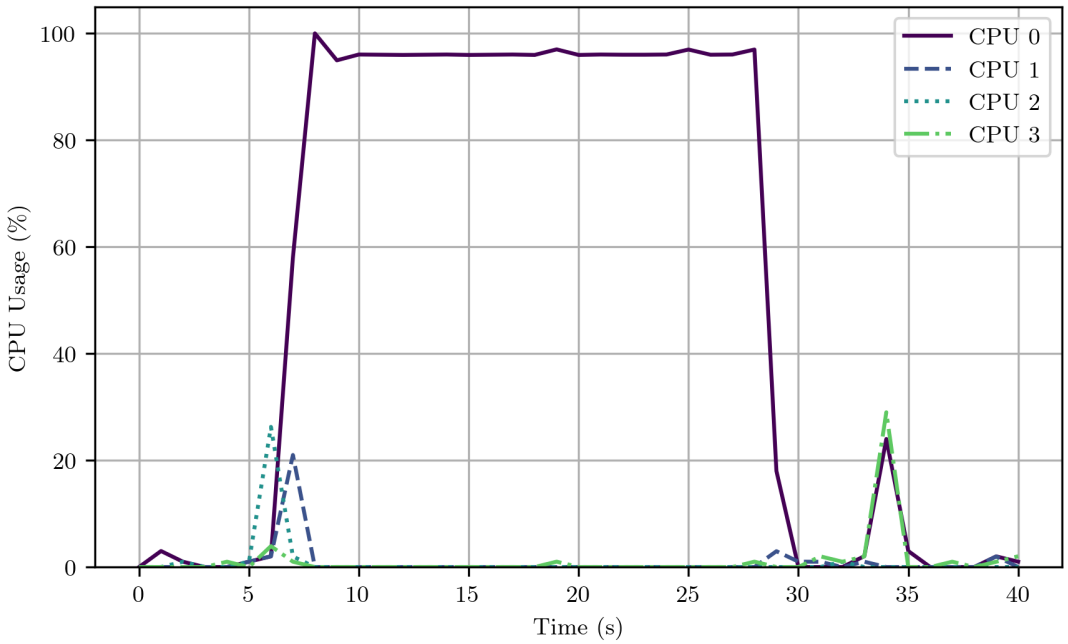

Figure A.20: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 1, T: 64)

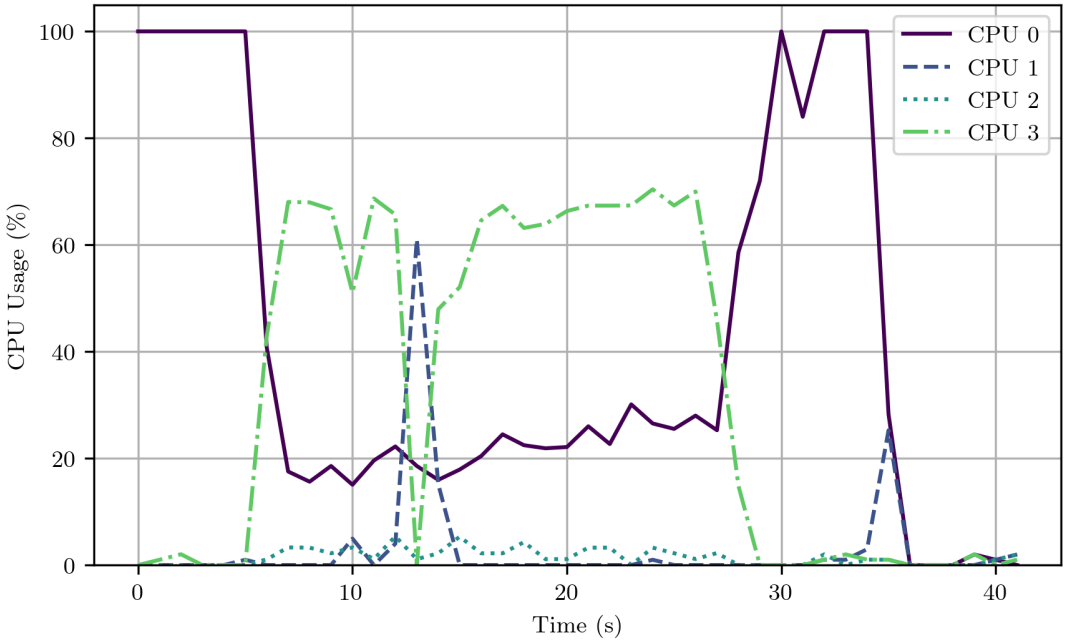

Figure A.21: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Proxy 1, Hops: 1, T: 64)

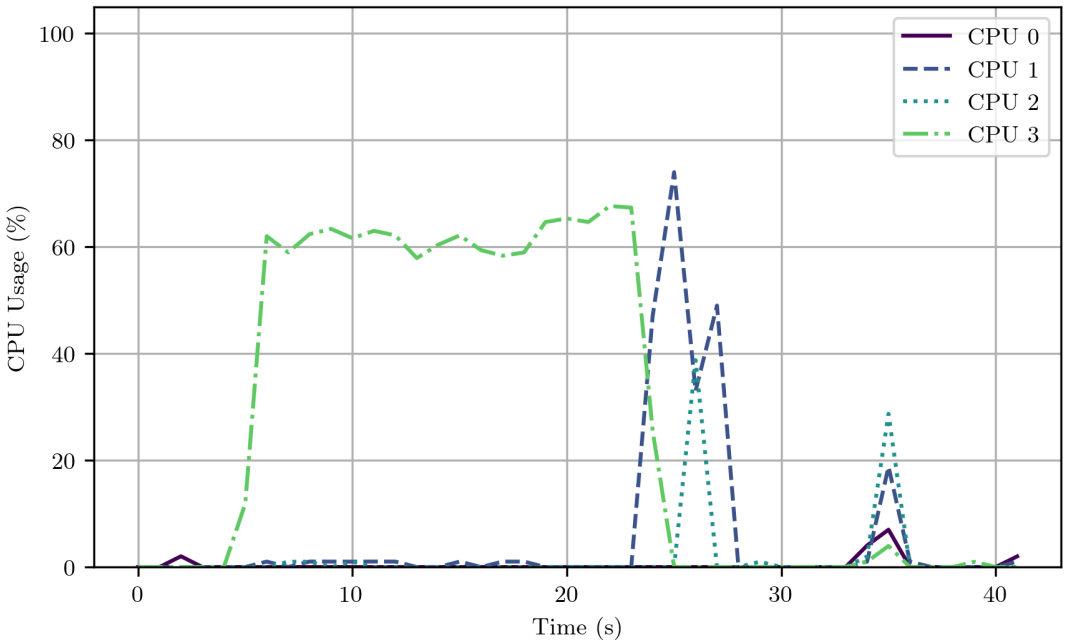

Figure A.22: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 1, T: 64)

A.2.3 Hops: 2

T: 1

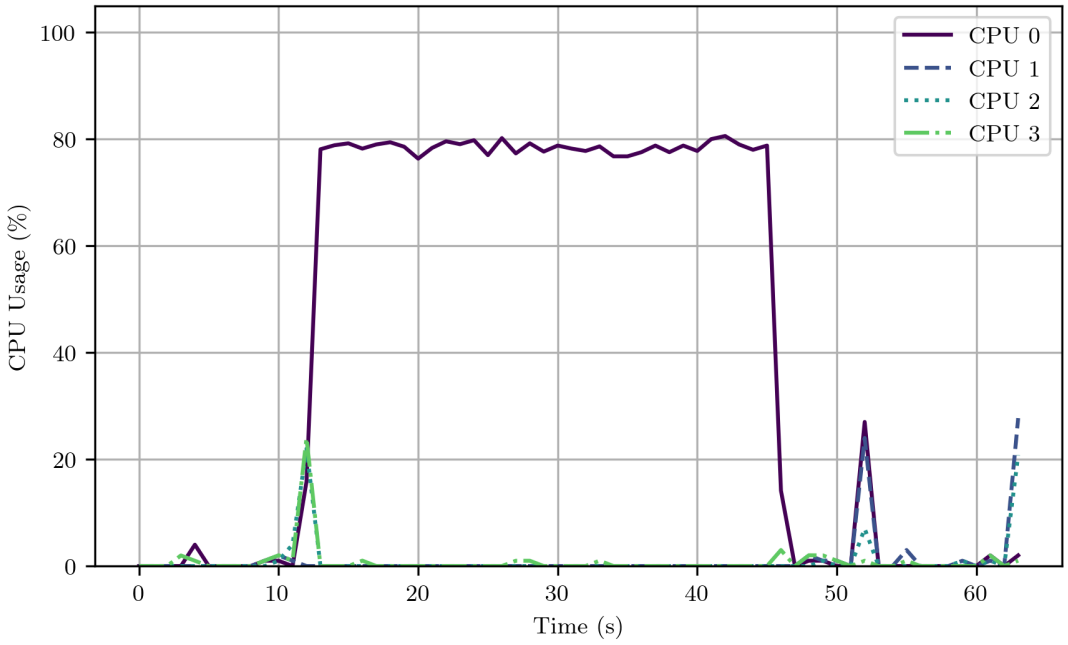

Figure A.23: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 2, T: 1)

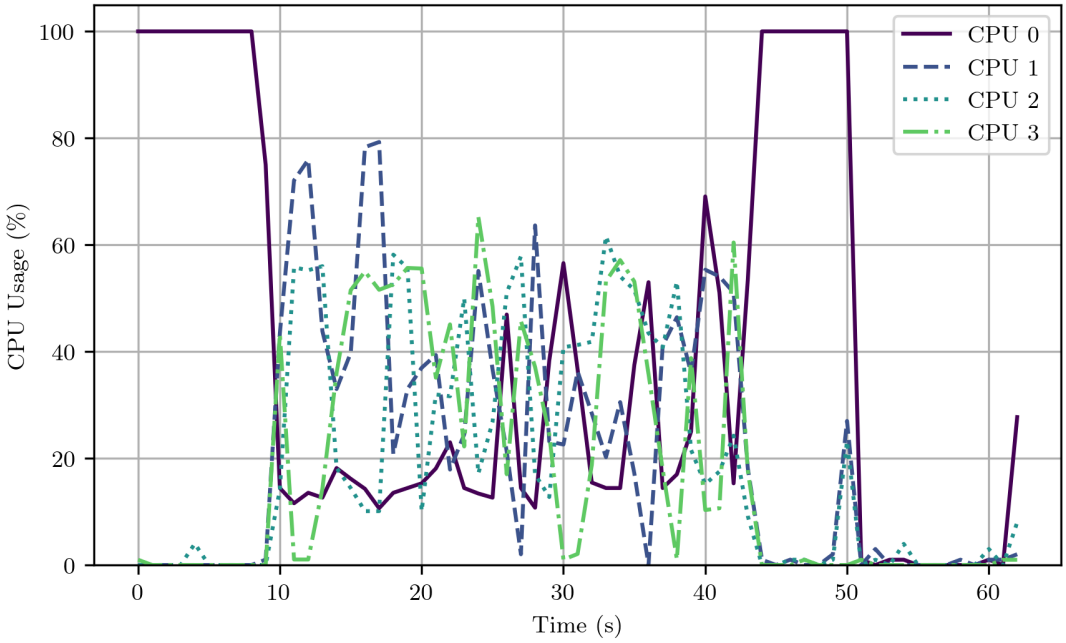

Figure A.24: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (1st Proxy, Hops: 2, T: 1)

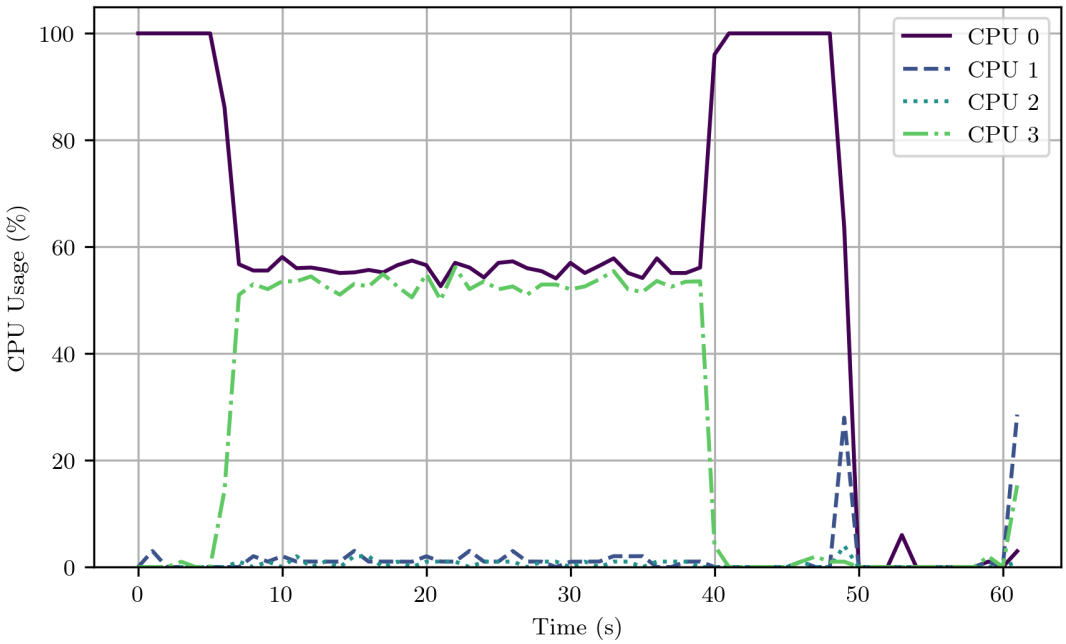

Figure A.25: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (2nd Proxy, Hops: 2, T: 1)

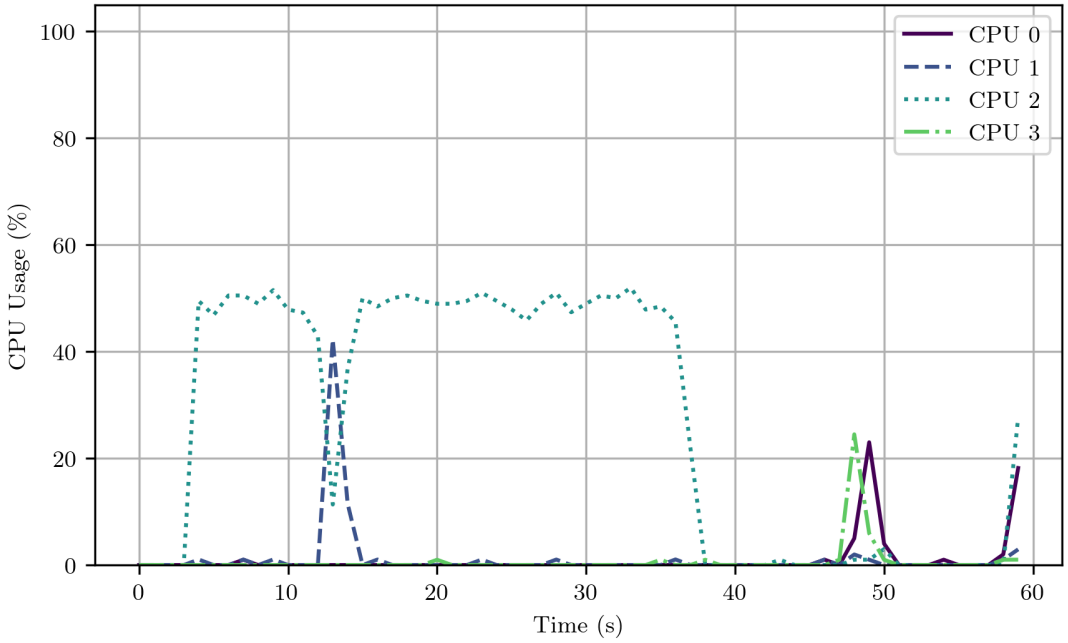

Figure A.26: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 2, T: 1)

T: 8

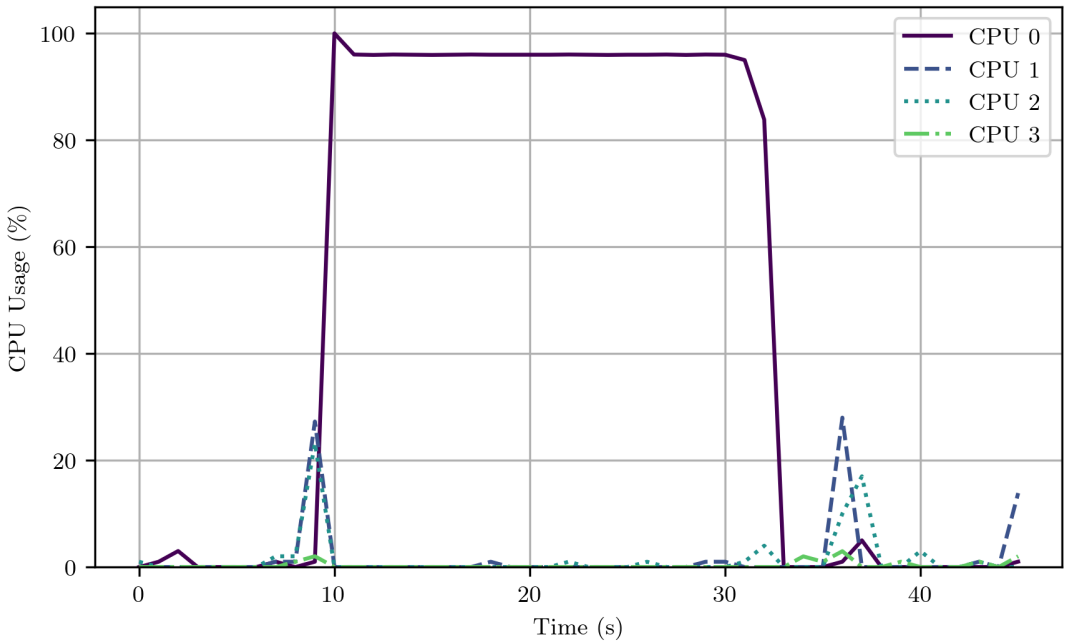

Figure A.27: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 2, T: 8)

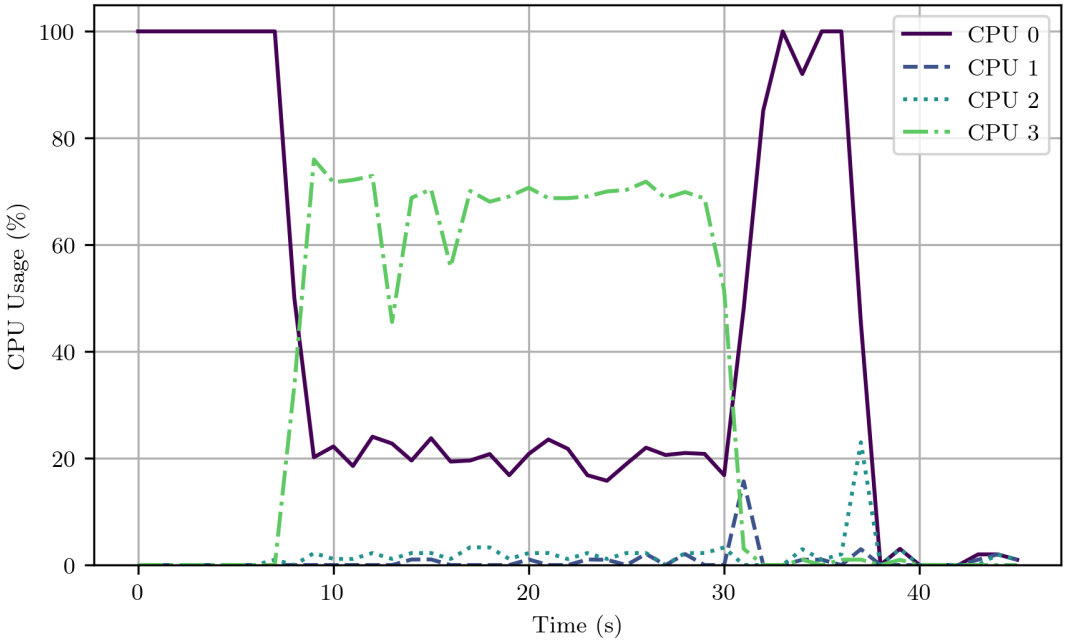

Figure A.28: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (1st Proxy, Hops: 2, T: 8)

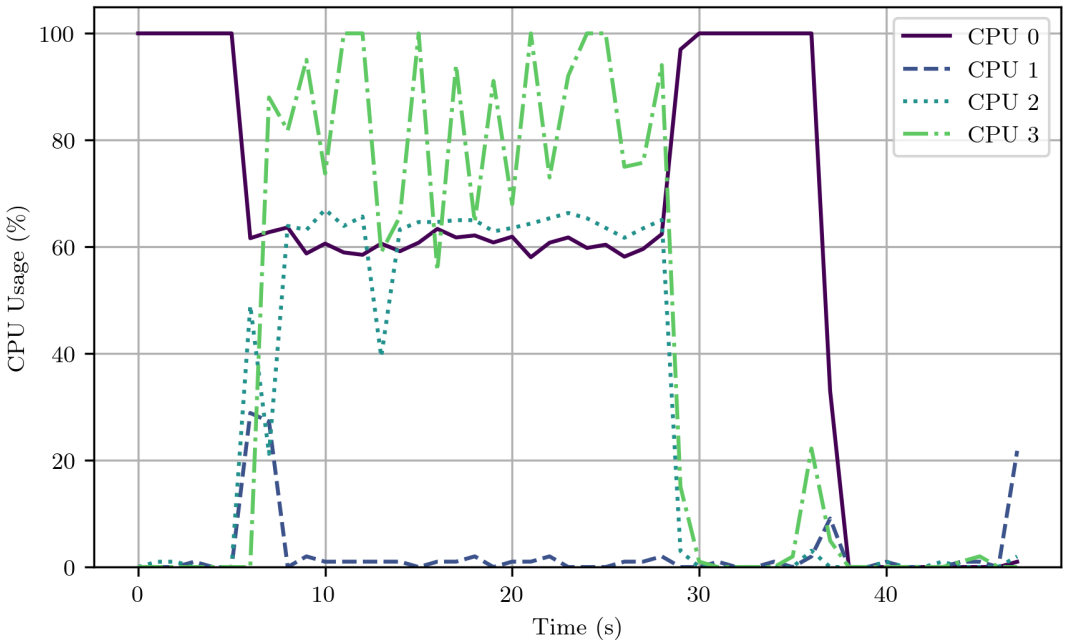

Figure A.29: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (2nd Proxy, Hops: 2, T: 8)

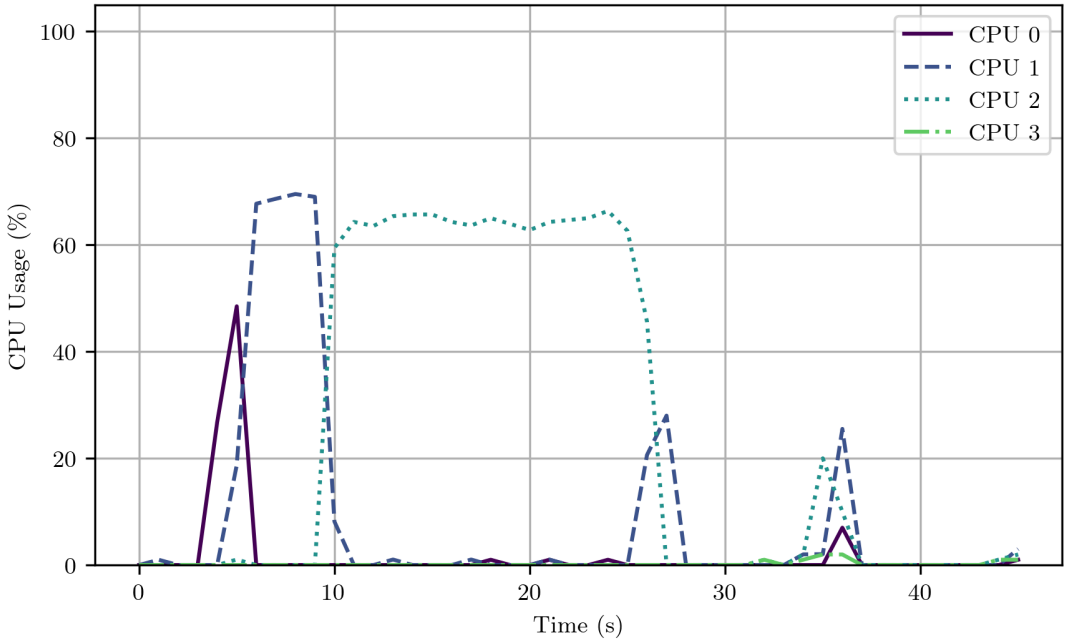

Figure A.30: HTTPConnectIP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 2, T: 8)

A.3 Selected CPU Usage for HTTPConnectUDP

A.3.1 Hops 4

T: 64

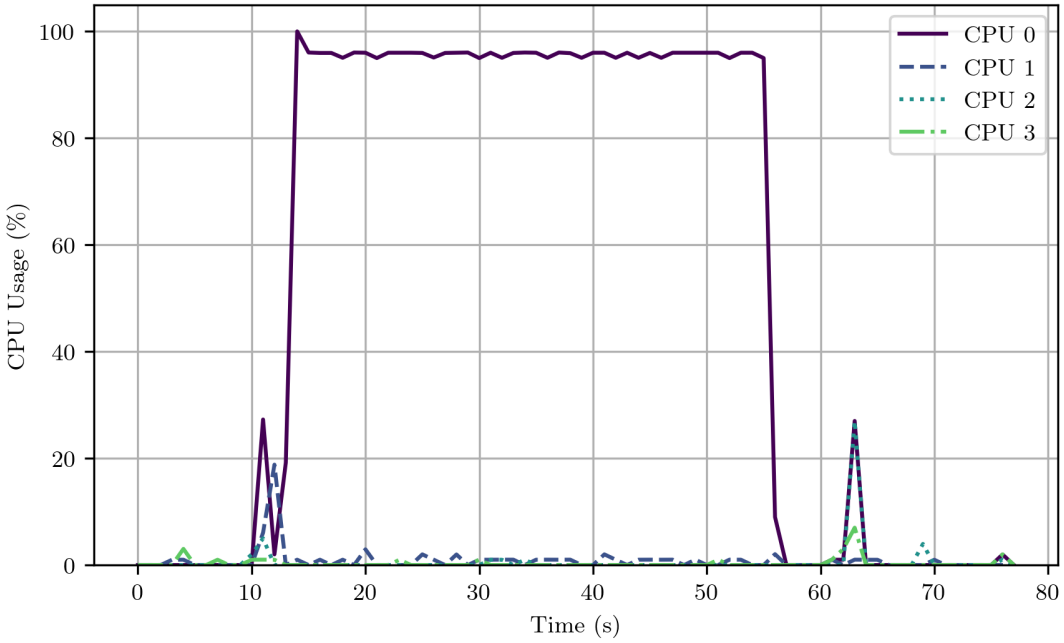

Figure A.31: HTTPConnectUDP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 4, T: 64)

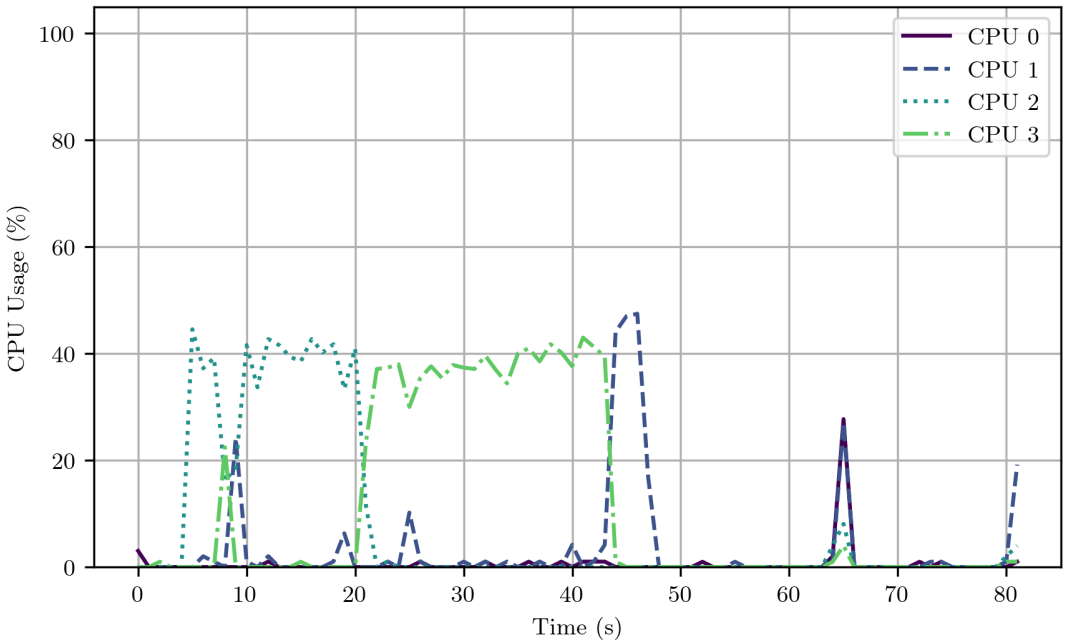

Figure A.32: HTTPConnectUDP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 4, T: 64)

T: 128

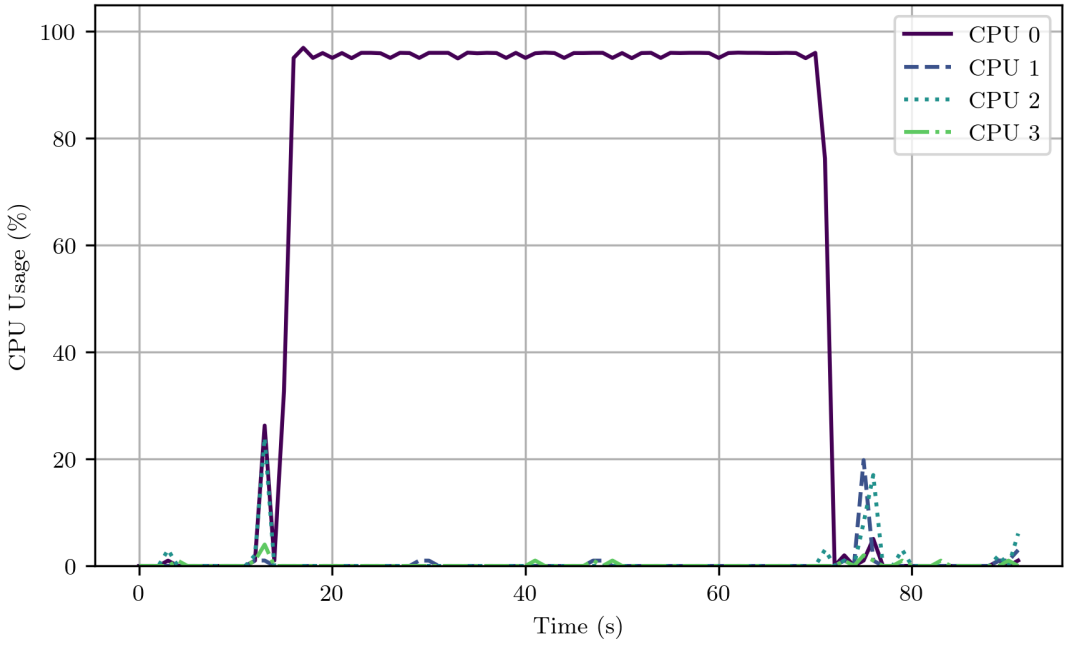

Figure A.33: HTTPConnectUDP: CPU Usage w/ Hyper-Threading disabled (Client, Hops: 4, T: 128)

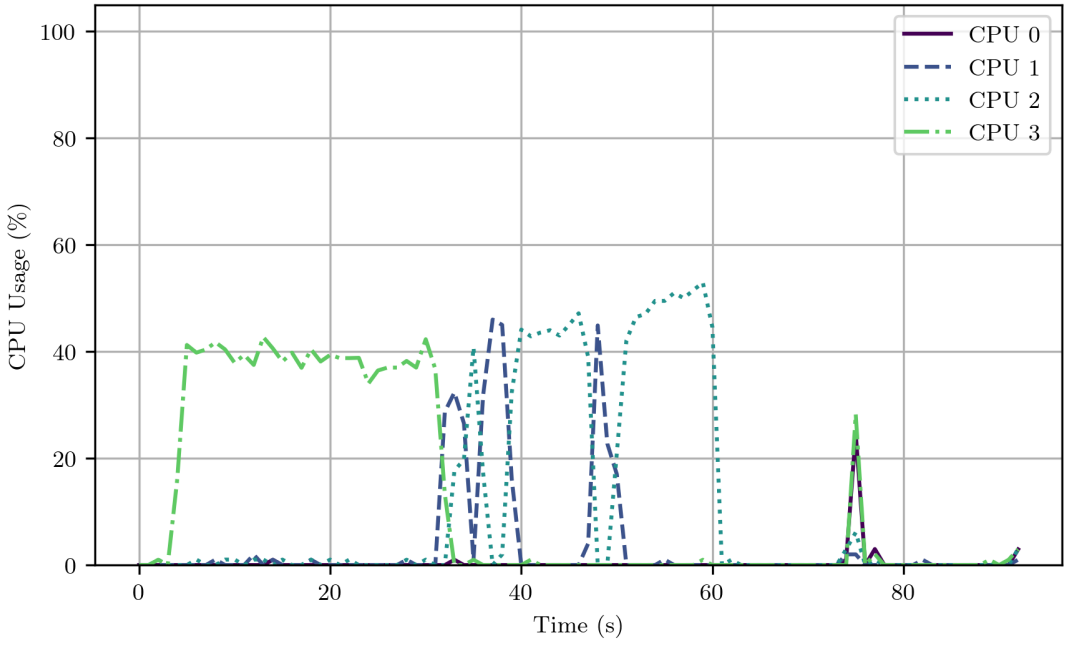

Figure A.34: HTTPConnectUDP: CPU Usage w/ Hyper-Threading disabled (Server, Hops: 4, T: 128)

Note that the respective proxy servers in these scenarios always had a CPU usage of 100%.

A.4 Selected CPU Usage for SeleniumConnectIP

A.4.1 Hops 0, Clients 1

T: 1

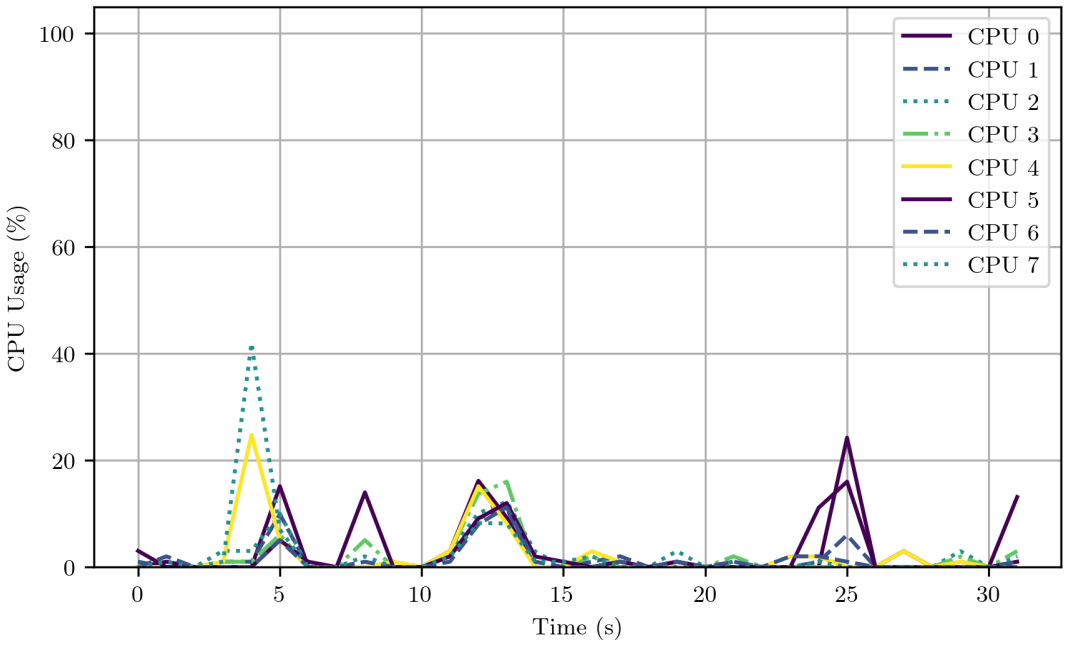

Figure A.35: SeleniumConnectIP: CPU Usage (Hops: 0, Clients 1, T: 1)

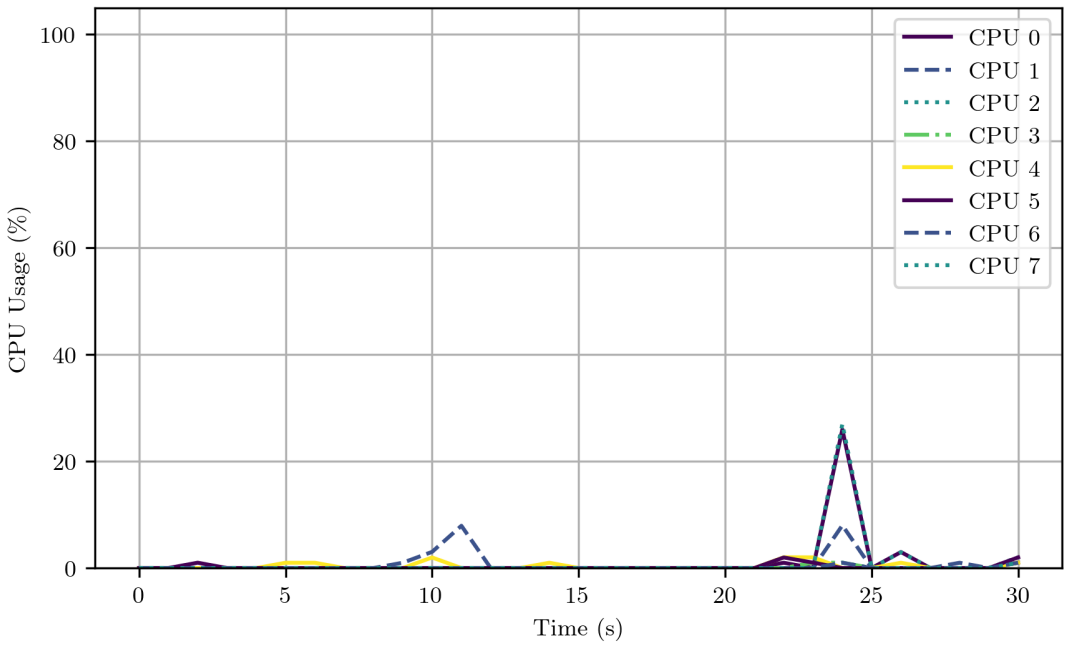

Figure A.36: SeleniumConnectIP: CPU Usage (Hops: 0, Server, T: 1)

T: 2

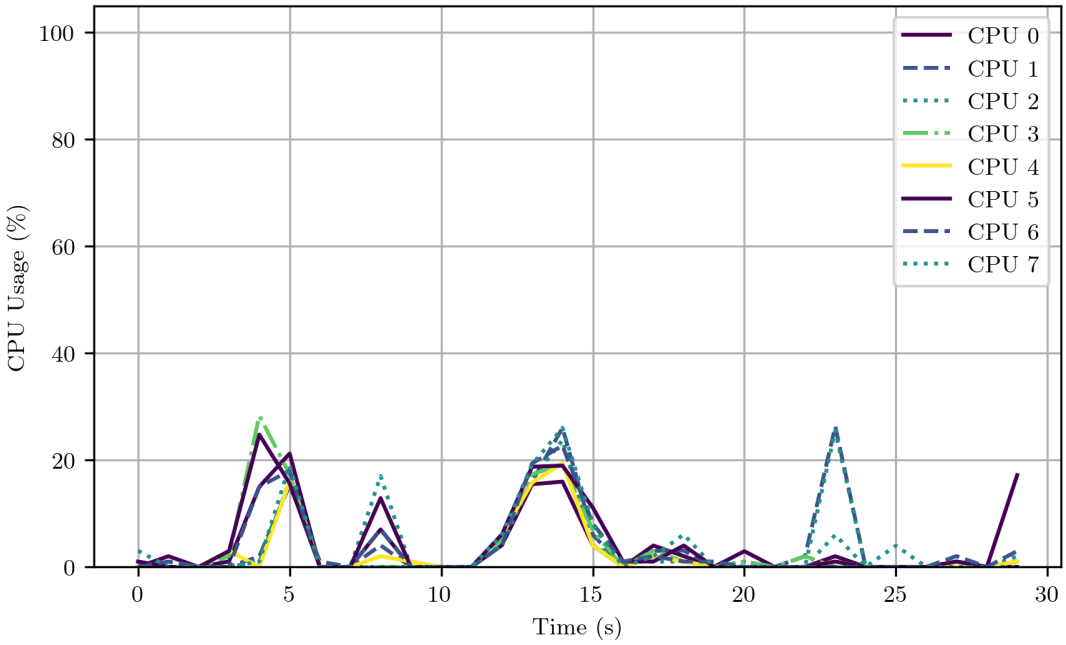

Figure A.37: SeleniumConnectIP: CPU Usage (Hops: 0, Client 1, T: 2)

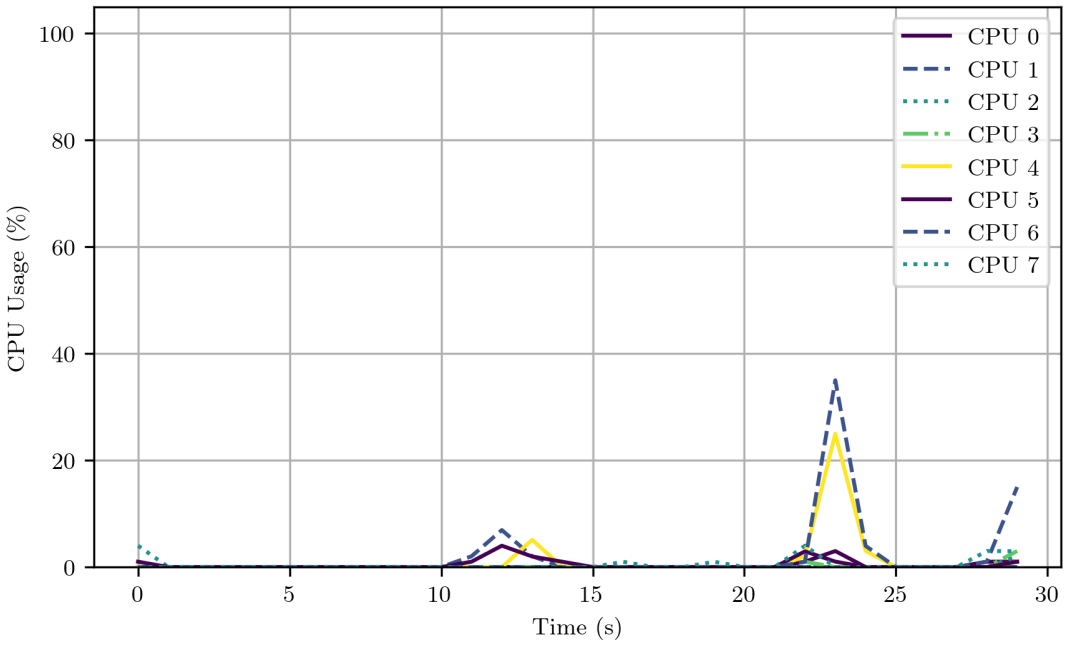

Figure A.38: SeleniumConnectIP: CPU Usage (Hops: 0, Server, T: 2)

T: 4

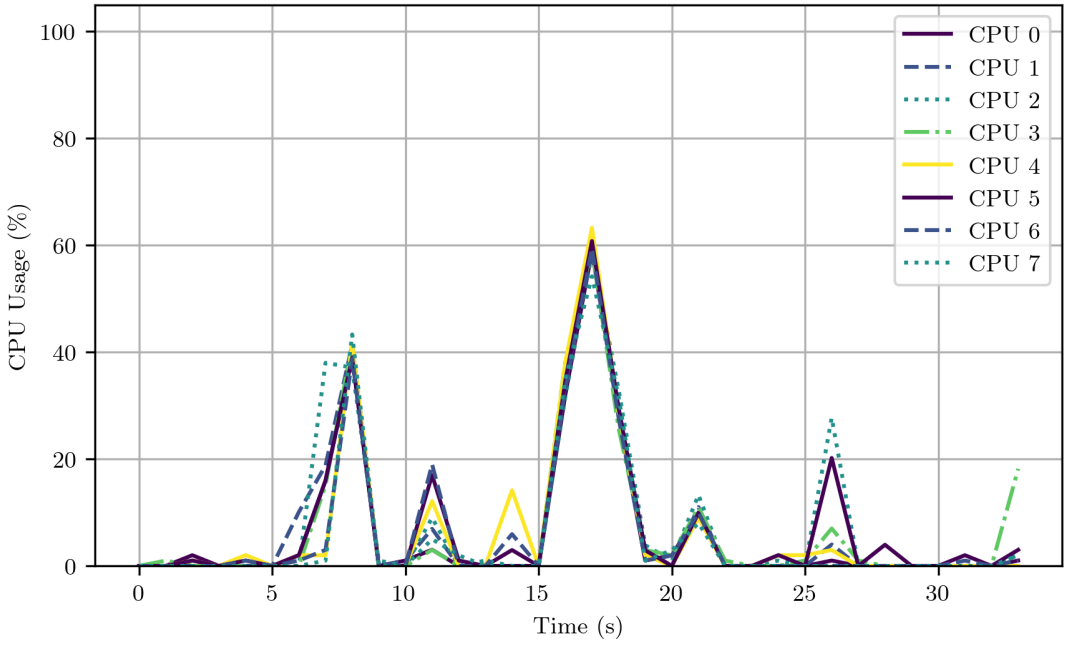

Figure A.39: SeleniumConnectIP: CPU Usage (Hops: 0, Client 1, T: 4)

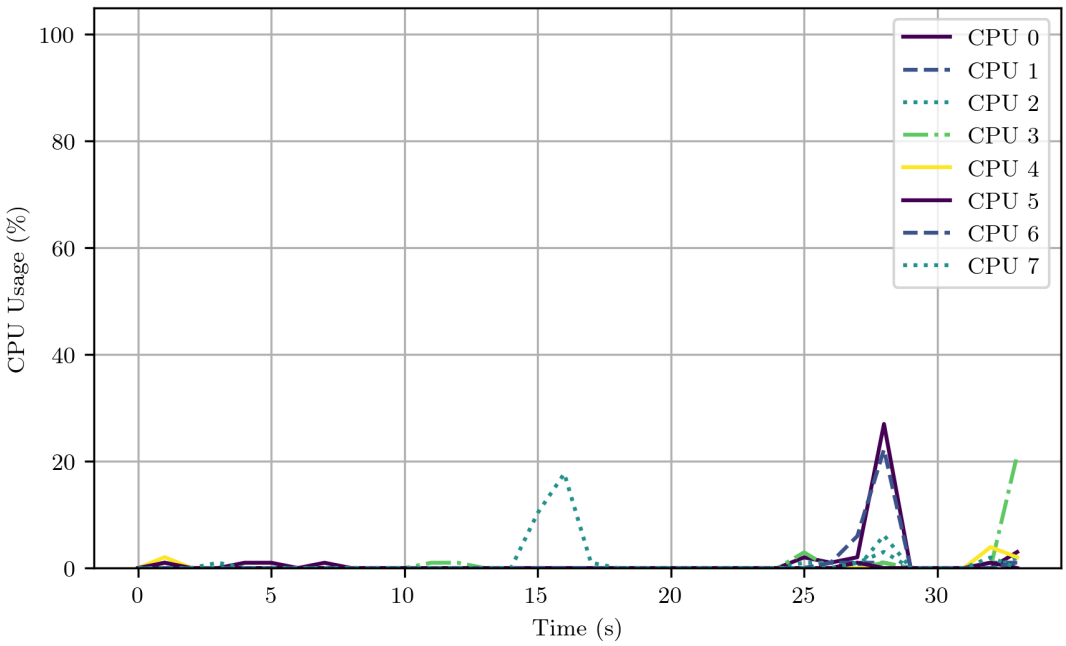

Figure A.40: SeleniumConnectIP: CPU Usage (Hops: 0, Server, T: 1)

T: 8

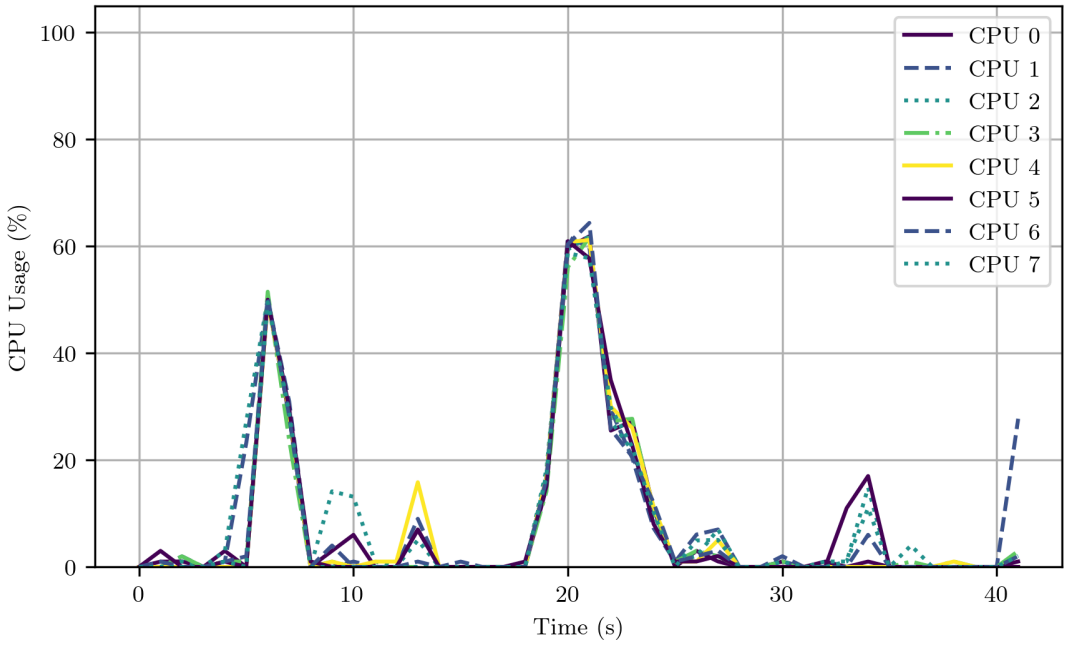

Figure A.41: SeleniumConnectIP: CPU Usage (Hops: 0, Client 1, T: 8)

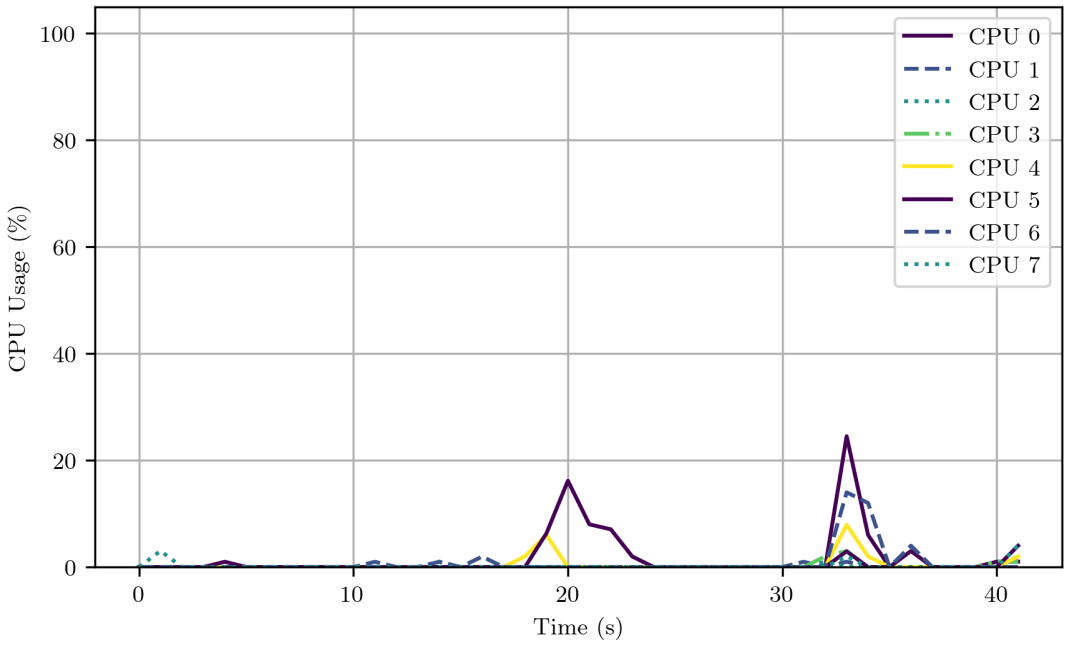

Figure A.42: SeleniumConnectIP: CPU Usage (Hops: 0, Server, T: 8)

Bibliography

[1] R. Clarke, "Risks inherent in the digital surveillance economy: A research agenda", *Journal of Information Technology*, vol. 34, no. 1, pp. 59–80, 2019. DOI: 10.1177/0268396218815559. eprint: https://doi.org/10.1177/0268396218815559. [Online]. Available: https://doi.org/10.1177/0268396218815559.

[2] D. Schinazi, *Proxying UDP in HTTP*, RFC 9298, Aug. 2022. DOI: 10.17487/RFC9298. [Online]. Available: https://www.rfc-editor.org/info/rfc9298.

[3] T. Pauly, D. Schinazi, A. Chernyakhovsky, M. Kühlewind, and M. Westerlund, "Proxying IP in HTTP", Internet Engineering Task Force, Internet-Draft draft-ietf-masque-connect-ip-13, Apr. 2023, Work in Progress, 41 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-masque-connect-ip/13/.

[4] *Multiplexed application substrate over quic encryption (masque)*. [Online]. Available: https://datatracker.ietf.org/wg/masque/documents/.

[5] R. T. Fielding, M. Nottingham, and J. Reschke, *HTTP Semantics*, RFC 9110, Jun. 2022. DOI: 10.17487/RFC9110. [Online]. Available: https://www.rfc-editor.org/info/rfc9110.

[6] *Ietf masque working group*. [Online]. Available: https://ietf-wg-masque.github.io/.

[7] T. Pauly, D. Schinazi, A. Chernyakhovsky, M. Kühlewind, and M. Westerlund, *Proxying IP in HTTP*, RFC 9484, Oct. 2023. DOI: 10.17487/RFC9484. [Online]. Available: https://www.rfc-editor.org/info/rfc9484.

[8] T. Pauly, D. Schinazi, A. Chernyakhovsky, M. Kühlewind, and M. Westerlund, "IP Proxying Support for HTTP", Internet Engineering Task Force, Internet-Draft draft-ietf-masque-connect-ip-00, Nov. 2021, Work in Progress, 20 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-masque-connect-ip/00/.

[9] M. D. Leech, *SOCKS Protocol Version 5*, RFC 1928, Mar. 1996. DOI: 10.17487/RFC1928. [Online]. Available: https://www.rfc-editor.org/info/rfc1928.

[10] R. Marx, *Head-of-line blocking in quic and http/3: The details*, 2020. [Online]. Available: https://calendar.perfplanet.com/2020/head-of-line-blocking-in-quic-and-http-3-the-details/.

[11] L. Pardue and C. Wood, *A primer on proxies*, 2022. [Online]. Available: https://blog.cloudflare.com/a-primer-on-proxies/.

[12] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, RFC 9000, May 2021. DOI: 10.17487/RFC9000. [Online]. Available: https://www.rfc-editor.org/info/rfc9000.

[13] M. Bishop, *HTTP/3*, RFC 9114, Jun. 2022. DOI: 10.17487/RFC9114. [Online]. Available: https://www.rfc-editor.org/info/rfc9114.

[14] T. Pauly, E. Kinnear, and D. Schinazi, *An Unreliable Datagram Extension to QUIC*, RFC 9221, Mar. 2022. DOI: 10.17487/RFC9221. [Online]. Available: https://www.rfc-editor.org/info/rfc9221.

[15] Apple, *Icloud private relay overview*, Accessed: 20.05.2023, 2021. [Online]. Available: https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf.

[16] R. Lalkaka, "Icloud private relay: Information for cloudflare customers", *The Cloudflare Blog*, 2022, Accessed: 20.05.2023. [Online]. Available: https://blog.cloudflare.com/icloud-private-relay/.

[17] R. Marx, *Head-of-line blocking in quic and http/3: The details*, 2020. [Online]. Available: https://calendar.perfplanet.com/2020/head-of-line-blocking-in-quic-and-http-3-the-details/.

[18] M. Kühlewind, M. Carlander-Reuterfelt, M. Ihlar, and M. Westerlund, "Evaluation of quic-based masque proxying", in *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*, ser. EPIQ '21, Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 2934, ISBN: 9781450391351. DOI: 10.1145/3488660.3493806. [Online]. Available: https://doi.org/10.1145/3488660.3493806.

[19] D. Scharnitzky, Z. Krämer, S. Molnár, and A. Mihály, "Real-time emulation of masque-based quic proxying in lte networks using ns-3",

[20] M. Kraft, "Local loss recovery and quic masque proxies to improve performance of satellite networks", Ph.D. dissertation, Jan. 2022.

[21] M. Galicer and C. Wood, *Privacy gateway: A privacy preserving proxy built on internet standards*, 2022. [Online]. Available: https://blog.cloudflare.com/building-privacy-into-internet-standards-and-how-to-make-your-app-more-private-today/.

[22] R. Marx, J. Herbots, W. Lamotte, and P. Quax, "Same standards, different decisions: A study of quic and http/3 implementation diversity", in *Proceedings of*

the Workshop on the Evolution, Performance, and Interoperability of QUIC, 2020, pp. 14–20.

[23] S. Hertelli, B. Jaeger, and J. Zirngibl, "Comparison of different quic implementations", *Network*, vol. 7, 2022.

[24] K. Ploch, "QUIC Performance on 10G Links", Bachelor's Thesis, Technical University of Munich, 2022.

[25] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, and G. Carle, "Quic on the highway: Evaluating performance on high-rate links",

[26] R. Marx, L. Niccolini, M. Seemann, and L. Pardue, "Main logging schema for qlog", Internet Engineering Task Force, Internet-Draft draft-ietf-quic-qlog-main-schema-06, Jul. 2023, Work in Progress, 50 pp. [Online]. Available: `https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/06/`.

[27] D. Schinazi, "UDP Proxying Support for HTTP", Internet Engineering Task Force, Internet-Draft draft-ietf-masque-connect-udp-07, Work in Progress, 16 pp. [Online]. Available: `https://datatracker.ietf.org/doc/draft-ietf-masque-connect-udp/07/`.

[28] M. Duke, "QUIC Version 2", Internet Engineering Task Force, Internet-Draft draft-ietf-quic-v2-10, Dec. 2022, Work in Progress, 18 pp. [Online]. Available: `https://datatracker.ietf.org/doc/draft-ietf-quic-v2/10/`.

[29] S. Dawaliby, A. Bradai, and Y. Pousset, "In depth performance evaluation of lte-m for m2m communications", in *2016 IEEE 12th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2016, pp. 1–8. DOI: `10.1109/WiMOB.2016.7763264`.

[30] V. Anand and D. Saxena, "Comparative study of modern web browsers based on their performance and evolution", in *2013 IEEE International Conference on Computational Intelligence and Computing Research*, 2013, pp. 1–5. DOI: `10.1109/ICCIC.2013.6724273`.

[31] M. Bünstorf and B. Jaeger, "Msquic–a high-speed quic implementation",