PEERPROXY: A WEBRTC PROXY FOR HTTP

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Nathan Lee

April 2025

COMMITTEE MEMBERSHIP

TITLE: PeerProxy: A WebRTC Proxy for HTTP

AUTHOR: Nathan Lee

DATE SUBMITTED: April 2025

COMMITTEE CHAIR: Joydeep Mukherjee, Ph.D.
Assistant Professor of Computer Science

COMMITTEE MEMBER: John Bellardo, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Ken Kubiak, Ph.D.
Lecturer

ABSTRACT

PeerProxy: A WebRTC Proxy for HTTP

Nathan Lee

Advances in networking technologies have empowered individuals to easily self-host digital services such as websites and smart home systems. However, accessing these services externally often requires port forwarding, which requires manual router configuration, technical expertise in networking, and is sometimes restricted by internet service providers. Proxy-based services such as Ngrok and Cloudflare Tunnels simplify external access by using publicly hosted proxy servers, but introduce increased infrastructure costs and privacy concerns due to reliance on third-party servers that can inspect or store traffic.

This thesis presents PeerProxy, a novel framework that simplifies access to self-hosted web services without manual network configuration, privacy risks, or increased infrastructure costs. PeerProxy is built on WebRTC, a set of protocols built into modern web browsers to form end-to-end encrypted peer-to-peer and proxied connections. This solution includes a custom local proxy server, a specialized browser client that loads web applications over WebRTC in unmodified browsers, and a lightweight signaling server for connection management. Additionally, this work introduces a custom packet protocol for efficiently transmitting HTTP messages over WebRTC data channels.

Performance evaluations show that while PeerProxy's download throughput (2.6 MB/s) is lower than traditional proxies (14.08 MB/s) due to WebRTC limitations, it maintains comparable latency for requests up to 1KB. Future proposed WebRTC improvements, such as RTCQuicTransport, could enhance its performance. Additionally, resource utilization tests confirmed that PeerProxy's browser client adds minimal overhead.

This research contributes a working prototype of PeerProxy, a thorough evaluation of HTTP proxying performance over WebRTC, and insights into the current limitations of high-throughput applications using modern browsers' implementations of WebRTC. It lays a foundation for further advancements in secure and decentralized web service hosting that empower users with secure, private, and efficient external access to self-hosted services.

Keywords: WebRTC, HTTP, Web Browsers, Self-Hosting, Proxies

ACKNOWLEDGMENTS

I would like to sincerely thank those who have supported and inspired me throughout this journey.

- I would like to express my deepest gratitude to my advisor, Dr. Mukherjee, for guiding me through the entire process of writing and refining my thesis. His insights into benchmarking and performance testing applications have been invaluable to this work.
- Thank you to Dr. Bellardo for his expert advice on networking topics and for validating many critical ideas throughout this project.
- Special thanks to Dr. Kubiak for his assistance with the browser-related aspects of this project and his invaluable guidance for implementing the browser client.
- I'd like to thank Sean Dubois for his work on the Pion project [85], the WebRTC library used in PeerProxy, for authoring WebRTC for the Curious [94], which provided essential technical insights, and his guidance on performance limitations of WebRTC for this project.
- I'd like to thank Rachel Izenson for her unwavering support and motivation during this lengthy and challenging journey.
- I also want to thank my family for their continuous support and for always encouraging me to be my best. Their unwavering belief in me and constant motivation have been essential through this journey.
- Finally, I would like to thank the faculty at Cal Poly throughout my journey through college, who have constantly inspired and encouraged me to think outside of the box.

Without all these people, this project would not have been possible, and I am profoundly grateful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation and Context

Advances in accessible technology have empowered individuals to host digital services directly from their homes. Some of these services include smart home systems and personal websites. While cloud services dominated the 2010s with their convenience and scalability [101], there has been a shift to self-hosting, where users are driven by concerns over privacy, data ownership, and service provider costs [110].

Although hosting services at home present technical challenges, these have been increasingly mitigated through widespread open-source self-hosting software and the adoption of containerization technologies. Open-source self-hosting software provides users with free, community-driven alternatives to commercial services. These projects simplify the process of setting up and maintaining services like media streaming, personal websites, and storage. Containerization, particularly with tools like Docker [29], further reduces technical barriers by allowing applications to run in isolated environments with all their dependencies included [7] which eliminates compatibility issues across different operating systems. With containerization, users can easily install, configure, and run self-hosted services without needing deep system administration knowledge.

A second significant barrier to self-hosting, and the primary focus of this thesis, is enabling external access to self-hosted services. The modern Internet primarily relies on centralized client-server architectures, where providers operate servers with dedicated networking infrastructure. Unlike home networks, these centralized services

typically utilize publicly accessible IP addresses, specialized routing, and sophisticated network configurations that facilitate direct, reliable external connectivity [73].

In contrast, most home networks are not designed for external access. Because devices on these networks are rarely intended to be publicly accessible, home networks commonly employ Network Address Translation (NAT), a router technique enabling multiple local devices to share a single public IP address [51]. While this improves security and conserves IPv4 addresses, it also complicates direct access from the external Internet because devices inside the network do not have publicly reachable addresses.

The typical way to allow direct access to a device in a network is to configure port forwarding on the router [51]. This is where a specific port on the router is configured to forward traffic to a designated device within the network. However, setting up port forwarding can be challenging for non-technical users, as it requires modifying router settings and understanding advanced networking topics [87]. Furthermore, numerous Internet Service Providers (ISPs) enforce restrictions such as carrier-grade NAT [108], making direct port-forwarding infeasible without advanced networking techniques.

Due to the complexities associated with direct access via port forwarding, proxy-based solutions have emerged to circumvent NAT restrictions and ISP limitations. Popular proxy-based solutions include Cloudflare Tunnels [19] and Ngrok [77]. These proxy-based solutions involve two components shown in figure 1.1. There's a publicly accessible hosted proxy server that accepts requests from external users. Next, there's a local proxy server, an application that runs on the same computer as the self-hosted service that forms a tunnel to the hosted proxy server. This setup allows external users to access a self-hosted service through an external proxy server without the need for port forwarding.

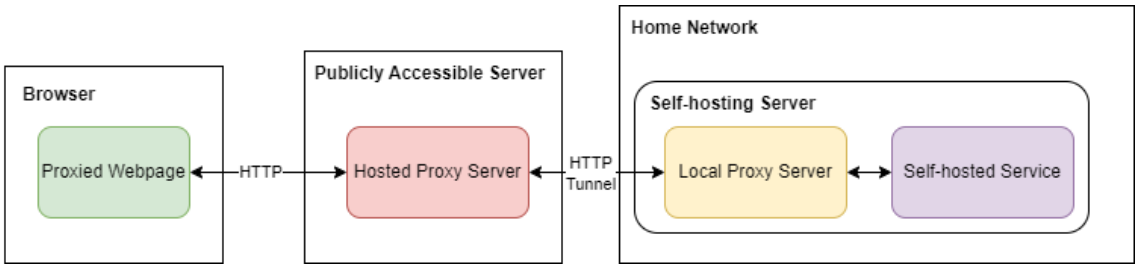


Figure 1.1: Proxy-based Local Service Hosting

While these proxy-based solutions eliminate the need for port forwarding, they introduce security concerns. Proxy servers inherently require Secure Sockets Layer/-Transport Layer Security (SSL/TLS) termination to function properly [48, 50]. This means that incoming encrypted traffic must be decrypted at the proxy server before being forwarded to its final destination, enabling efficient routing but introducing security concerns but breaking end-to-end encryption [78]. Users have to place trust in the proxy provider since they have the technical capability to inspect or modify traffic passing through their infrastructure. Additionally, if a proxy provider is compromised, attackers could gain access to all traffic passing through it, creating substantial security risks [40].

Additionally, building and maintaining infrastructure for a proxy service can be expensive and complicated, and the cost can be passed down to the user. The amount of infrastructure needed scales with the user traffic and can become quite expensive. Additionally, to give users the best experience with minimal latency and good throughput, proxy servers must be close to the user and the device to which they are trying to connect. To achieve this, a service provider must build or pay for a globally distributed network of proxy servers, further increasing service providers' costs.

1.2 Proposed Solution

This thesis proposes PeerProxy, a framework that enables seamless communication with web services behind a local network without manual network configuration or unencrypted communication through third-party proxies. PeerProxy leverages WebRTC, a peer-to-peer communication protocol designed for real-time data exchange [89]. It uses a combination of techniques to form direct connections between peers behind local networks. It falls back to end-to-end encrypted communication through a proxy server if a direct connection isn't possible.

Since all communication with WebRTC is end-to-end encrypted, this mitigates the risk of a third party inspecting and modifying communication, returning full ownership and control to users. Furthermore, PeerProxy significantly reduces infrastructure costs for service providers by reducing the need for dedicated relay servers. WebRTC's design ensures that only a small portion of traffic has to be sent through relay servers. Based on usage data gathered by appear.net, a live streaming company, 82% of WebRTC connections are made directly peer-to-peer [49]. This could translate into an 82% reduction in relaying infrastructure costs for users and service providers. By providing a direct, secure, and efficient communication system, PeerProxy addresses the privacy concerns of end users and the operational costs of service providers.

1.2.1 PeerProxy Implementation

PeerProxy addresses the same problem as proxy-based solutions, accessing self-hosted services behind home networks, but PeerProxy uses WebRTC as the transport for enhanced privacy and efficiency. Similar to how proxy-based solutions work, there is a local proxy server on the same machine as the self-hosted service. However,

instead of forming a tunnel with a hosted proxy server, it directly connects to a user's browser or forms an encrypted tunnel through a proxy server. While browsers natively support WebRTC [35], they do not support it as a transport for serving a website, so PeerProxy has a custom browser client that loads and displays a self-hosted website. These components are shown in figure 1.2.

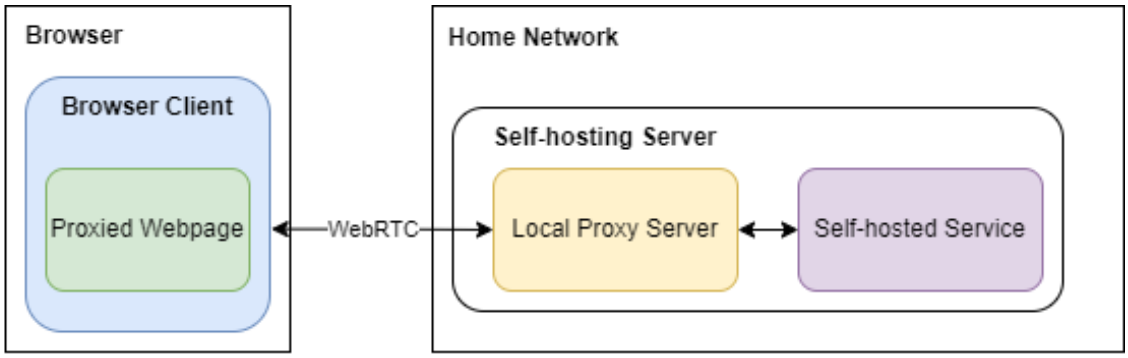


Figure 1.2: High Level Overview of PeerProxy

The PeerProxy interface is designed to be as easy to use as proxy-based services like Ngrok or Cloudflare tunnels. To make a self-hosted service accessible over the Internet using PeerProxy, the user first needs to start the PeerProxy local proxy server on the same network or device as the self-hosted service.

For example, if the web server in a local network were running on port 3000, the user would run the following command to start the local proxy server:

```
$ peerproxy 3000
```

Then, PeerProxy would provide the user with a unique URL, such as:

```
https://3fdw.peerproxy.dev
```

The user can then open the URL on any device, regardless of network location, and view the self-hosted service.

1.3 Research Questions and Answers

RQ1: How can support for loading a webpage over a WebRTC connection be added in unmodified modern browsers?

WebRTC connections are supported in browsers but not as a means of transport for loading websites. This thesis demonstrates that it is possible to load web content over WebRTC through the use of a specialized browser client. The browser client is a web application that leverages existing web APIs to load and render a website. The design of the browser client makes it so a website is displayed and functions as if it was loaded normally. The browser client allows unmodified modern browsers, such as Chrome, Safari, and Firefox, to load a web application over a WebRTC-based transport seamlessly.

RQ2: What is an efficient way to send HTTP messages through a WebRTC data channel that minimizes extraneous data sent while preserving reliability?

This thesis proposes a custom packet protocol tailored for WebRTC data channels, the primary API for sending data over a WebRTC connection [69]. Since data channels handle data as discrete packets, HTTP messages are serialized into smaller chunks that fit within the data channel's recommended packet size. Each packet includes carefully designed metadata, such as sequence numbers and packet type flags, to ensure proper ordering and complete reassembly at the destination. Since it's a custom packet protocol, the metadata is stored efficiently, minimizing overhead while preserving the reliability of HTTP messages.

RQ3: How does PeerProxy compare in terms of throughput, latency, and resource usage to existing commercial proxies?

This work includes extensive evaluations comparing performance trade-offs between using a proxy-based solution and PeerProxy. The primary trade-off identified is throughput. For example, download tests showed that PeerProxy reached only 2.6 MB/s compared to Ngrok's 14.08 MB/s. Meanwhile, latency tests showed that for small payloads (up to around 1KB), PeerProxy's performance is similar to Ngrok's.

A significant finding of this work is that PeerProxy's performance limitations primarily stem from how browsers implement WebRTC. WebRTC uses the SCTP (Stream Control Transmission Protocol) network protocol to transmit data, a protocol similar to TCP. Unlike TCP, which automatically adjusts the amount of data it sends based on network conditions, the SCTP implementation in most browsers uses a fixed limit. This means it doesn't scale to handle more significant amounts of data efficiently, creating a throughput bottleneck under heavy load or longer delays. Although the current SCTP implementation in browsers doesn't scale efficiently, this doesn't mean that browsers can't eventually achieve performance comparable to TCP. There are proposals like the QUIC API for peer-to-peer connections [107] that explore alternative transports with higher performance.

Despite this limitation, PeerProxy can achieve comparable or even better throughput than proxy-based solutions in scenarios with suboptimal regional deployment. For instance, when a user is located in the same region as the self-hosted service, but the proxy server is hosted in a distant region, the resulting routing will travel back and forth from that distant region, reducing performance. This scenario is common if a service provider doesn't have infrastructure in a certain region. PeerProxy instead can use the shortest path from the user to the self-hosted service. In a download

throughput test, PeerProxy achieved 2.6 MB/s, while a proxy-based solution with a suboptimal proxy region only achieved 1.2 MB/s.

Additionally, PeerProxy incurs an initial latency overhead from establishing the WebRTC connection, averaging around 482.8 ms under optimal conditions, potentially exceeding 1100 ms in realistic scenarios. However, with some optimization techniques in the browser client to reuse WebRTC connections across page navigations, this initial overhead is only for the initial page load.

Furthermore, PeerProxy incurs slightly increased CPU and memory usage within browsers, primarily because its browser client operates in user space rather than utilizing the browser's optimized network stack. In a high workload benchmark, PeerProxy used 5.63% CPU compared to a commercial proxy using 1.96% and a 5% increase in memory usage. However, this increased resource usage is negligible in modern devices.

1.4 Thesis Organization

- **Background and Related Works**: Provides background information on WebRTC, HTTP, and the various web technologies PeerProxy uses. This chapter also reviews related work, including existing WebRTC performance evaluation in the real world and some similar applications of WebRTC.
- **Implementation**: This chapter is a deep dive into how various components work together in PeerProxy.
- **Experimental Setup**: This chapter explains the various benchmarks for evaluating PeerProxy.

- **Results and Evaluation**: Presents the results of the experiments, diving deep into throughput, latency, and resource usage benchmark results. This chapter also explores the causes of limitations of WebRTC in modern browsers.
- **Conclusion and Future Work**: Concludes the thesis, briefly discusses future work, and summarizes key contributions and findings.

Chapter 2

BACKGROUND

2.1 Background

The following chapter provides an overview of what WebRTC is and how it works, as well as an overview of some of the other web technologies used in this thesis.

2.1.1 WebRTC Overview

Web Real-Time Communication (WebRTC) is an open standard that enables two peers to communicate in real-time [89] [3]. WebRTC supports transmitting video, voice, and generic data, allowing developers to build powerful video and voice solutions. All major browsers support WebRTC [35], and there are various client libraries for server and mobile environments.

WebRTC supports direct peer-to-peer communication between clients without intermediate servers. This has a few benefits over traditional client-server communication. Transmitting data peer-to-peer leads to reduced bandwidth costs since developers don't have to host separate servers to relay media. This also has security benefits since users aren't routing data through third-party servers and don't have to trust that the servers aren't decrypting or reading any data [28]. The following subsections expand more on WebRTC.

2.1.2 WebRTC History

Google released the WebRTC project in May 2011 to bring a standard set of APIs for audio and voice to browsers [2]. This new API was created to replace the cluttered landscape of browser plugins made with Adobe Flash or Java Applets. These plugins had security vulnerabilities and compatibility issues and required users to install additional software. In October 2011, the W3C (World Wide Web Consortium) published a first draft of the protocol [106]. According to MDN (Mozilla Developer Network), WebRTC has been "baseline widely available" in browsers since January 2020 [35], meaning it's widely supported and safe to use. During the global pandemic in 2020, people had to find new ways to work, educate, and connect with others via video chat. Suddenly, WebRTC became a crucial web technology, allowing billions of individuals and organizations to continue their daily activities remotely. Chrome saw a 100-fold increase in received video streams with WebRTC in 2020 [14].

2.1.3 Connecting Two Peers With WebRTC

In real-world networking, connecting two peers directly is a difficult problem. Modern internet networking involves routers, NATs, firewalls, and VPNs that complicate direct connections. WebRTC was developed to tackle the challenges of peer-to-peer connections. It includes a subsystem called ICE (Internet Connectivity Establishment) dedicated to directly connecting two peers. The below outlines portions of what is covered in the book WebRTC for the Curious [94]. The following section goes over different networking situations that WebRTC has to account for.

The first scenario is the simplest, involving two devices on the same network, as shown in figure 2.1. Each device has its own private IP address assigned by the router. It's simple for Devices 1 and 2 to connect using their private IP addresses since they're in the same networking segment.

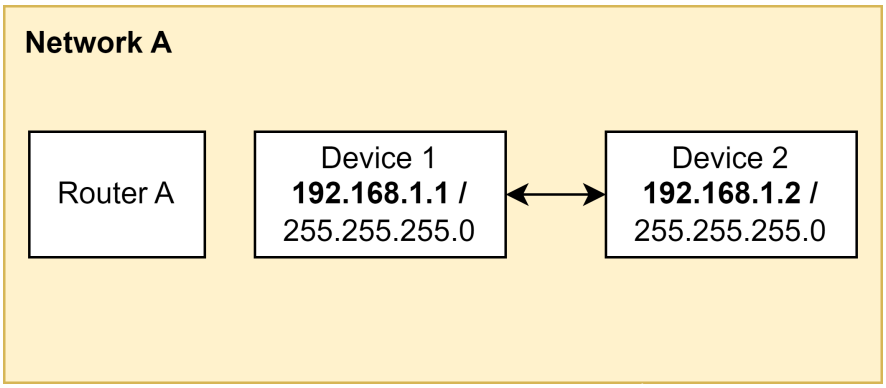


Figure 2.1: Simple Networking Setup

However, it's a trickier scenario if two devices on different networks want to connect directly peer-to-peer. In figure 2.2, if Device 1 (192.168.1.1) wants to communicate with Device 3 (also 192.168.1.1), they cannot simply use private IP addresses because their IP addresses are not unique across different networks and are not routable on the public internet.

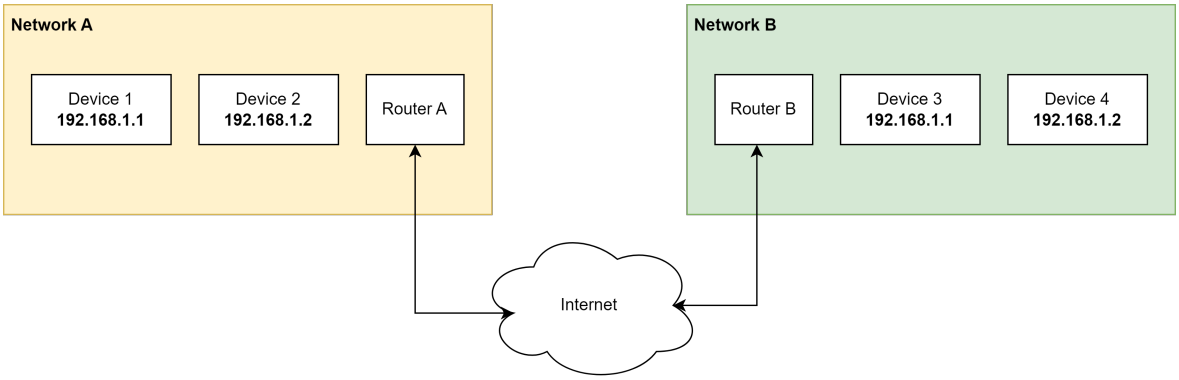


Figure 2.2: Typical Networking Setup

Since you can't connect two peers on different networks simply by exchanging private IP addresses, WebRTC relies on a different mechanism, figuring out a device's NAT (Network Address Translation) mapping.

2.1.3.1 Determining NAT Mapping

NAT (Network Address Translation) translates an IP address space into another by modifying IP packets as they go through a routing device [51]. NAT mapping was developed as a solution to a shortage of IPv4 addresses in the early 1990s. IPv4 uses 32-bit addresses, so it can only uniquely address 4.4 billion devices.

NAT mapping allows a router to have a single public IP address and track multiple devices with private IP addresses. It accomplishes this by assigning a temporary public IP and port to a device or request. Then, when data is sent back to that temporary IP and port, the router knows which device to send it back to. The following is an example of how a router uses NAT to manage requests from multiple devices under one public IP address. This is shown in figure 2.3.

1. Device 1 sends a request to an external server at 2.0.0.0.
2. Router A intercepts that request and assigns the request to port 5000 on the router's public IP address. It records that mapping in the NAT translation table. There are different types of NATs, and the behaviors are discussed in section 2.1.3.3.
3. All IP packets have a source and destination IP address [53]. The "source" field in the packet is changed from the original IP address to the router-assigned mapping.
4. The request is sent to the original destination.

5. Since the source field was changed, the external server sends the packet to the router-assigned mapping. When the router receives a response packet, it looks up which local device was assigned to that destination in the NAT table.

6. The router forwards that packet to the original device.

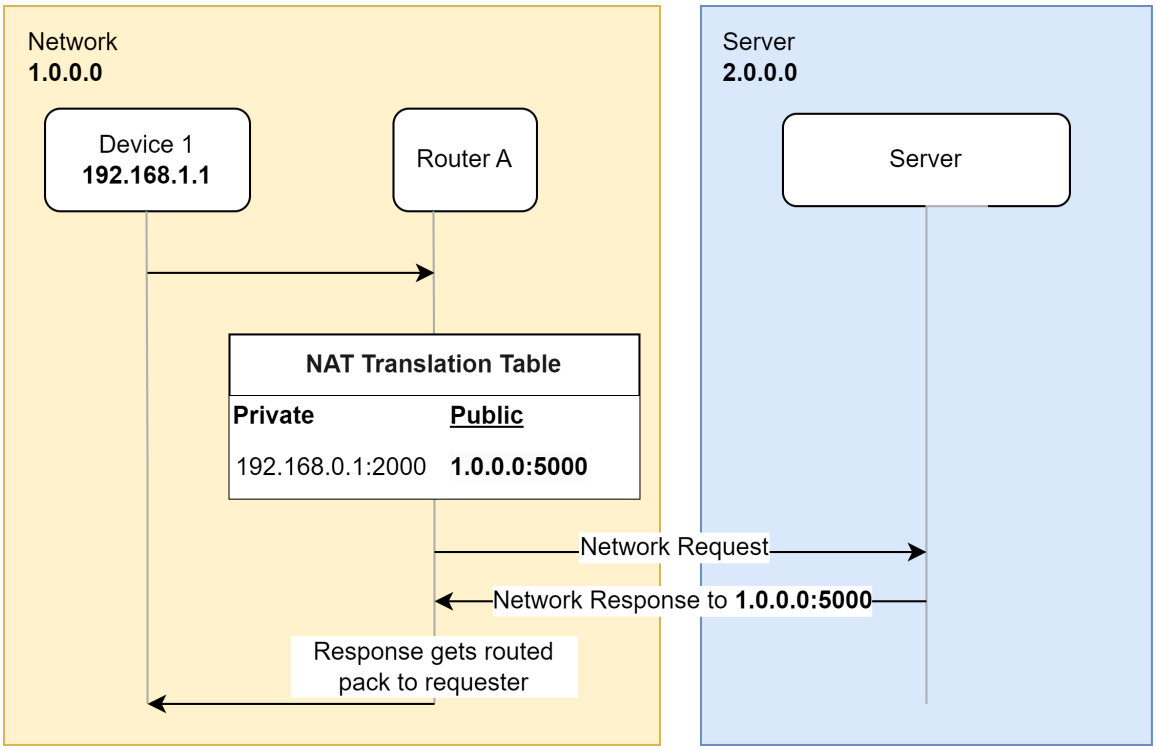


Figure 2.3: NAT Translation

2.1.3.2 STUN: Session Traversal Utilities for NAT

If two peers on different networks know each other's NAT-assigned IP and port, they can communicate directly. However, a device inside a network doesn't know its NAT mapping since the router handles it. WebRTC uses STUN to programmatically create and reflect the NAT mapping for the original device. STUN is defined in RFC 8489 [83].

The STUN process uses an external publicly hosted STUN server. This process can give the device its NAT mapping with a single request, as shown in figure 2.4.

1. Device 1 sends a *STUN Binding Request* to the STUN server.
2. The router modifies the outgoing request by changing the source of the packet to a temporary IP address and port.
3. The STUN Server receives the request and reads the source of the packet. Then, it puts the source into the body of the packet, the *STUN Binding Response*.
4. Device 1 receives the request and now knows its NAT mapping.

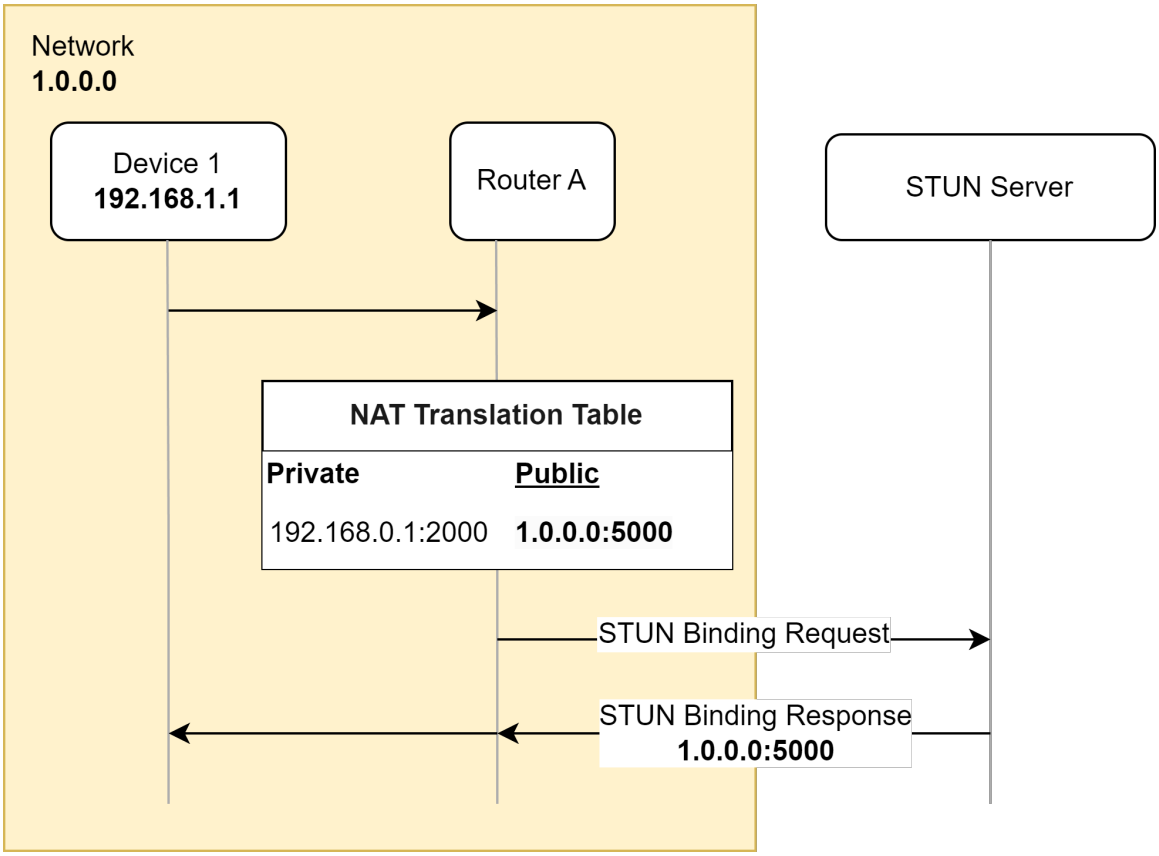


Figure 2.4: STUN

An external peer can then use that NAT mapping to attempt to form a direct connection. However, since there is no direct connection yet, there needs to be a way for peers to pass each other connection information. In WebRTC, this is known as signaling. In most implementations, this is an intermediate server that both peers can send messages through. Signaling is further explored in section 2.1.4.

The signaling and direct connection process is shown in figure 2.5.

1. The two peers determine their NAT mapped addresses with STUN. Then, they send the addresses to each other with an intermediate signaling server.

2. Peer 2 sends data to Peer 1 through Peer 1's mapped address and visa versa.

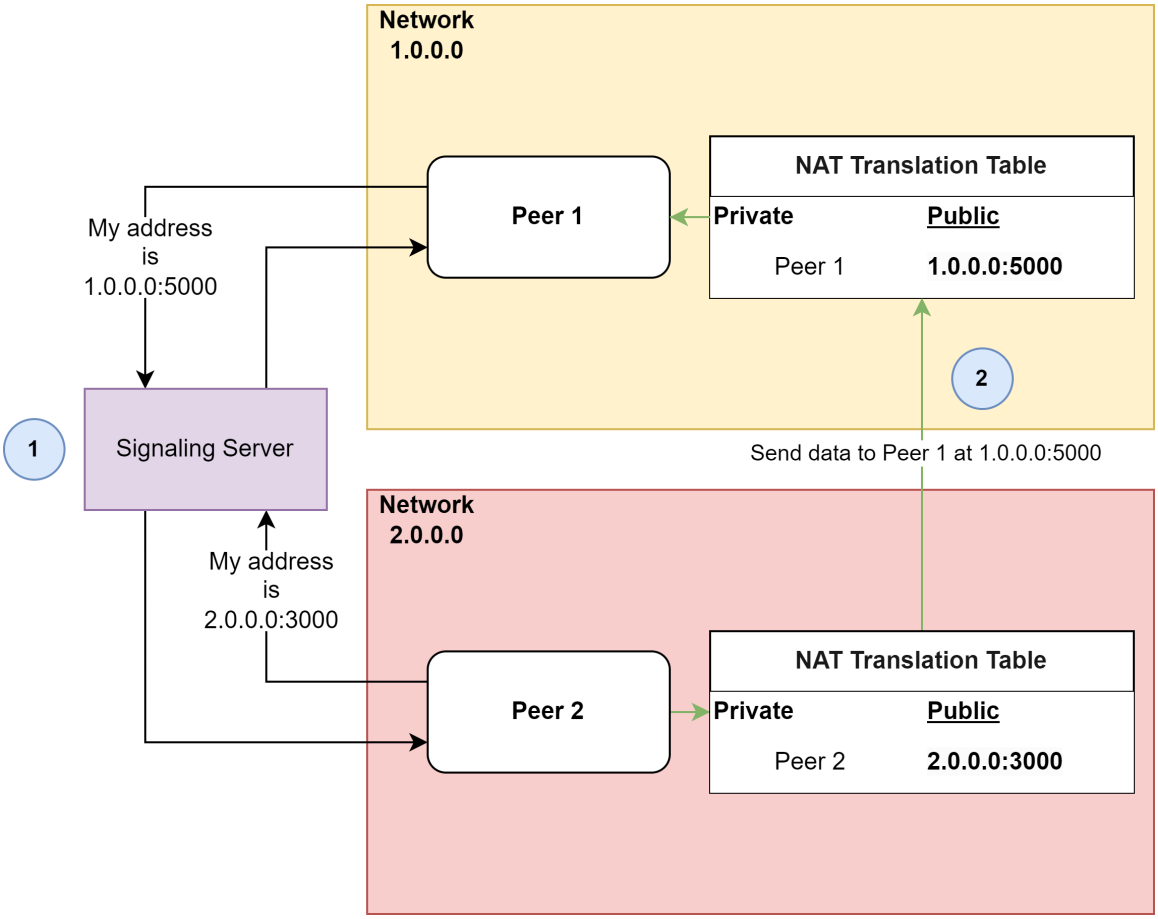


Figure 2.5: Connecting two peers with STUN

2.1.3.3 Types of NAT Mapping

The NAT traversal example discussed in section 2.1.3.2 is a simplified version of how NATs behave in the real world. The NATs shown in that example are full cone NATs which don't have any restrictions on incoming data, but some NATs restrict who can send data to the binding. There are 2 main types of NATs that handle creating mappings differently: cone and symmetric NATs [93].

Cone NATs: With a cone NAT [93], every request a device makes uses the same mapped address. That means that an external peer can send messages to that mapped address, which will be routed back to the address. There are three variations of Cone NATs, each differing in how they handle incoming traffic:

- **Full Cone:** any external peer can send packets to the mapped address
- **Restricted Cone:** only accepts packets from the same external IP address the internal device has already contacted
- **Port-Restricted Cone:** only allows incoming packets from the same external IP address and port that the internal device previously contacted.

Cone NATs are the easiest to form connections with since each device reuses the same public address mapping. Direct peer-to-peer connections with WebRTC require at least one peer to have a Cone NAT, which will be discussed later in this section.

Symmetric NATs: With a symmetric NAT [93], each outgoing connection request from a device is mapped to a unique external port and IP address. That means that the only external devices that can connect to the device are those it has contacted first. This restriction makes symmetric NATs more difficult to connect to. A direct

peer-to-peer connection is impossible if both peers have a symmetric NAT. This is outlined below and accompanied by figure 2.6.

1. The peers exchange mapped addresses obtained through STUN. Since the peers are behind the symmetric NAT, they cannot receive any data through that mapped address except from that STUN server.

2. Peer 1 attempts to send a packet to Peer 2's mapped address.

3. However, Peer 2's mapped address from STUN can only receive data from the STUN server. Any packets from Peer 1 are blocked.

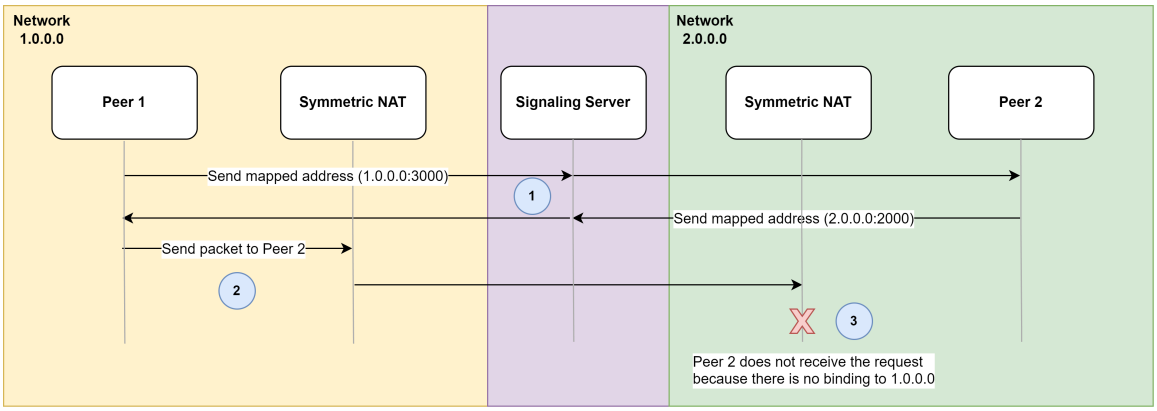


Figure 2.6: Connecting two peers directly behind symmetric NATs is more difficult

If a direct connection isn't possible, both peers have to communicate with an intermediate proxy server with a protocol called TURN (Traversing Using Relays around NAT), further discussed in section 2.1.3.4.

However, a direct connection is possible when one peer is behind a symmetric NAT and the other is behind a full cone or restricted cone NAT. An example of a symmetric NAT peer and restricted cone NAT peer connecting is outlined below and shown in

figure 2.7. The same applies to a full cone NAT. A full table of which types of NATs can form direct connections is shown in table 2.1.

In this example, peer 1 is behind a symmetric NAT, and peer 2 is behind a port restricted cone NAT.

1. The peers exchange mapped addresses obtained through STUN with a signaling server.
2. Peer 2 attempts to send a packet to peer 1 at 1.0.0.0:3000. The cone NAT reuses the same public address but allows requests from 1.0.0.0.
3. Peer 1's symmetric NAT drops the packet since there is no binding for request from 2.0.0.0 yet.
4. Peer 1 sends a packet to the Peer 2 address obtained through STUN. Since Peer 1 is behind a symmetric NAT, a new NAT binding is created at 1.0.0.0:4000. Only Peer 2 can send data through that binding.
5. From step 2, Peer 2's mapped address accepts packets from 1.0.0.0.
6. Peer 2 can then send data to Peer 1 because the new NAT binding at Peer 1's side permits incoming packets from Peer 2's public address and port.
7. Peer 1 receives those packets, and bidirectional communication is established.

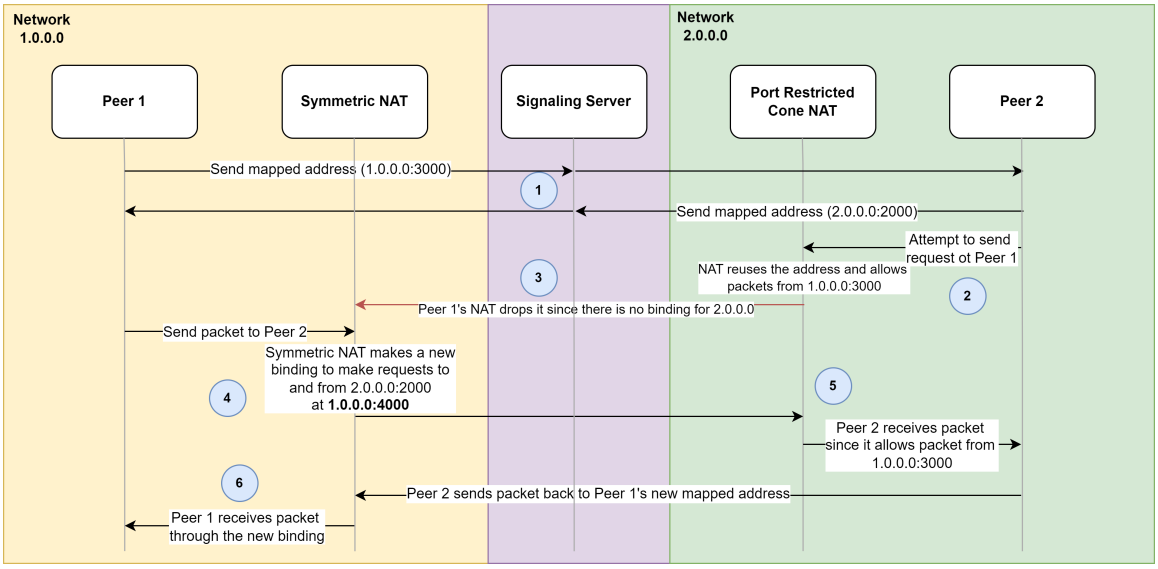


Figure 2.7: Connection is possible if one peer is behind a cone NAT

Table 2.1 shows an overview of which combinations of NATs can directly connect with STUN or require an intermediate TURN server. In the real world, based on usage data gathered by appear.net, 18% of WebRTC connections require a TURN server.

Table 2.1: STUN and TURN Requirements Based on NAT Types [97]

	Full Cone	**Restricted Cone**	**Port Restricted Cone**	**Symmetric**
Full Cone	STUN	STUN	STUN	STUN
Restricted Cone	STUN	STUN	STUN	STUN
Port Restricted Cone	STUN	STUN	STUN	TURN
Symmetric	STUN	STUN	TURN	TURN

2.1.3.4 TURN (Traversing Using Relays around NAT)

TURN is defined in RFC 8656 [90] and is the fallback solution when a direct connection isn't possible, such as when both peers are behind symmetric NATs. TURN uses a dedicated server that acts as a proxy for a client, enabling communication between peers that would otherwise be unable to form a direct connection.

In order to use TURN, a peer makes a request to the server to create a TURN allocation. This allocation provisions a tunnel for other peers to the original peer. Once the TURN server makes the allocation, it returns a "Relay Transport Address". Another peer can send data to the relay transport address, and the data will be forwarded to the original peer. This relay transport address is sent to peers during signaling as a possible path to send data to the original peer. This process is shown in figure 2.8.

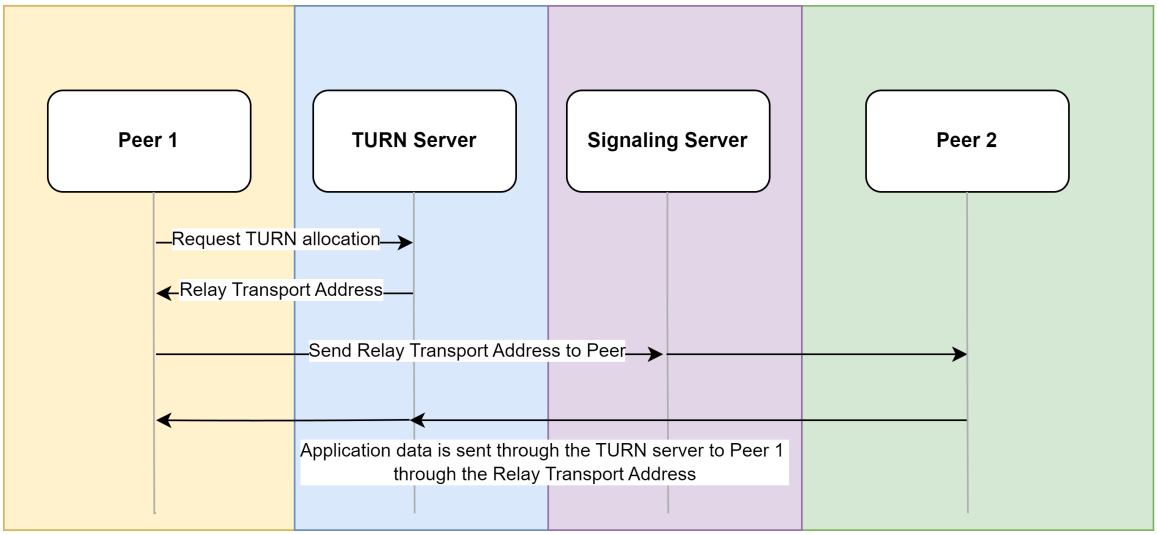


Figure 2.8: TURN allocation

This sounds similar to how commercial reverse proxies work, but it has some advantages. TURN is a fallback for when a direct connection isn't possible, and most network situations don't require it. With WebRTC, traffic going through TURN

servers is end-to-end encrypted. Even if a TURN server is compromised, an attacker wouldn't be able to access or manipulate the traffic.

2.1.3.5 ICE (Interactive Connectivity Establishment)

ICE is a protocol that bridges all the aforementioned connection techniques and scenarios together (Same Network, STUN, TURN) [59]. Every WebRTC client gathers a set of potential connection points. For example, a WebRTC client would determine their local IP address, STUN binding, and TURN allocation address. These addresses are known as ICE candidates. As these are discovered, those candidates are sent to the other peer through an external communication system known as signaling, discussed in section 2.1.4.

These are the types of ICE candidates:

Host: The locally assigned IP address and port. A host candidate would be used if two peers were on the same network.

mDNS (Multicast DNS): mDNS candidates are a more private version of host candidates [17]. It replaces the local IP address with a generated hostname ending in ".local". This prevents exposing a user's local IP address during ICE candidate generation. First, the peer generates a hostname ending in ".local". The connecting peer receives that hostname through signaling and sends a multicast DNS query on the local network for that hostname. The original peer then responds with its local IP address and the two peers can connect directly.

Server Reflexive: A server reflexive candidate is generated through the STUN process. The candidate is the NAT-mapped address. This candidate allows two peers to connect directly across different networks, as discussed in section 2.1.3.2.

Relay: This is a relayed transport address allocated by the TURN server, an intermediate server used to connect two peers when a direct connection isn't possible. This is discussed in section 2.1.3.4. This relayed transport address can be used to send data to the peer through the TURN server.

Once the candidates are exchanged, the ICE agent attempts to find a pair of candidates that work. Peer 1 will try to send packets through the host, server reflexive, and relay candidates of Peer 2. Peer 2 will try the same for Peer 1's candidates. The candidate pairs do not have to be the same type of candidates. For example, one peer could be behind a restrictive network and need a relay candidate to receive data, while the other is behind a more open network and needs a server reflexive candidate to receive data. This is shown in figure 2.9, where an ICE agent can send data to the other peer's candidates regardless of how it gets data.

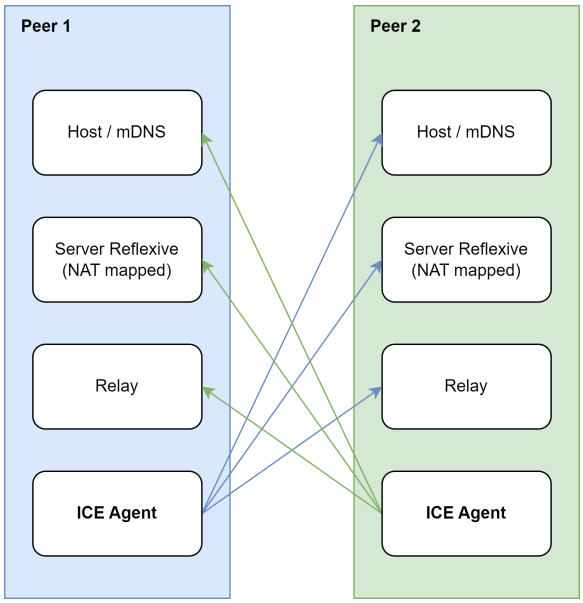


Figure 2.9: Potential ICE Candidate Pairs

Once the peers find a connection that works, that candidate pair is elected and used for all traffic moving forward. ICE bridges many protocols and techniques which are summarized in table 2.2.

Table 2.2: WebRTC Connection Acronyms and Descriptions

Term	Description
WebRTC	A set of protocols that create direct peer-to-peer communication between two clients
ICE	A technique to find ways for two peers to connect directly to each other
NAT	Translates an IP address space into another by modifying IP packets as they go through a routing device
STUN	A protocol to allow a peer to determine its own NAT mapping by sending a request to a STUN server
TURN	If two peers can't form a direct connection, traffic is routed through an external TURN server

2.1.3.6 SDP: Session Description Protocol

Along with determining a route to connect two peers, the peers need to negotiate media formats, encryption settings, and network configuration to establish a compatible connection. SDP or Session description protocol (defined in RFC 8866 [11]) is how two peers exchange information before establishing a connection. SDP is structured as a series of key-value pairs describing various session parameters.

WebRTC uses SDP for an offer-and-answer framework. One peer creates an "offer" that describes its media capabilities and DTLS encryption keys (section 2.1.6). The other peer reads that offer and creates an answer that confirms or rejects different media capabilities.

2.1.4 Signaling: Exchanging ICE Candidates

In order to connect two peers directly, the ICE candidates have to be exchanged between the two peers using a signaling server beforehand. WebRTC doesn't specify any specific protocol or transport for signaling or exchanging connection information [38]. The signaling server has to be accessible by both peers and forward information between them. Most WebRTC projects use a WebSocket server (WebSockets are described in more detail in section 2.2) since they allow for bidirectional and efficient message exchange.

Signaling usually involves exchanging three types of messages: offer, answer, and ICE candidates. Offer and answer messages contain SDP descriptions (described in section 2.1.3.6). ICE candidates contain any type of candidate listed in section 2.1.3.5.

These messages are exchanged in the following order, also shown in figure 2.10.

1. Peer 1 generates an SDP offer and sends it to Peer 2 through the signaling server
2. Peer 2 receives the offer and generates an answer.
3. Peer 2 sends the offer to Peer 1 through the signaling server.
4. Both peers send ICE candidates as they discover them (e.g., making STUN requests, getting a transport address in a TURN server, etc.).

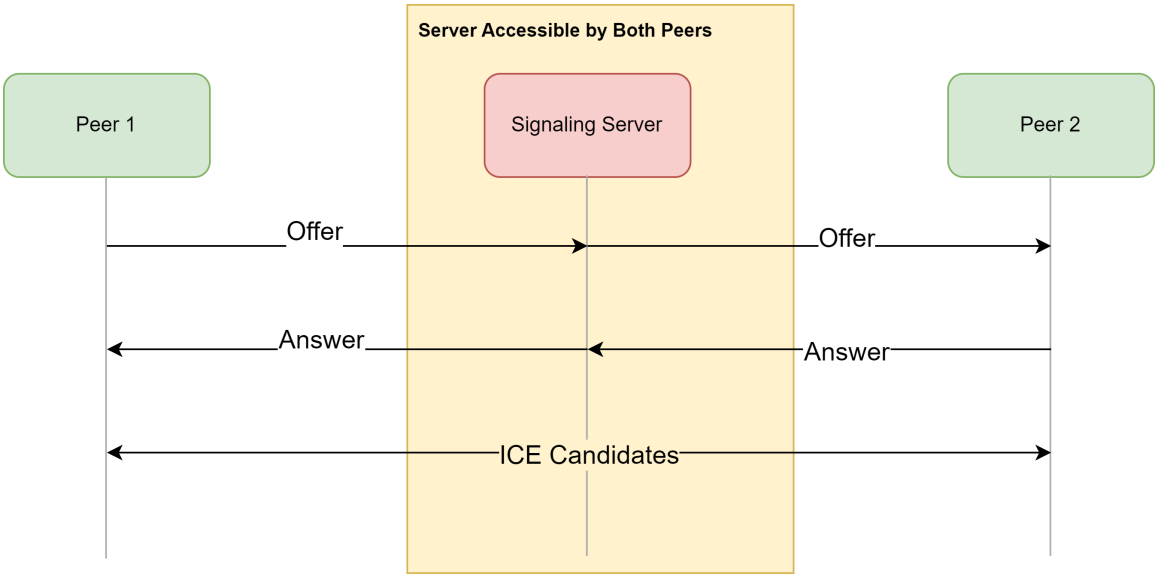


Figure 2.10: WebRTC signaling sequence

A timeline of the signaling messages sent and received in a WebRTC connection is shown in figure 2.11. The offer and answers are exchanged first in "SDP Exchange". The offer and answers contain some ICE candidates, and they are immediately added to each peer. Then, as each peer discovers ICE candidates, they get sent. Finally, after testing ICE pairs, a connection is finally formed.

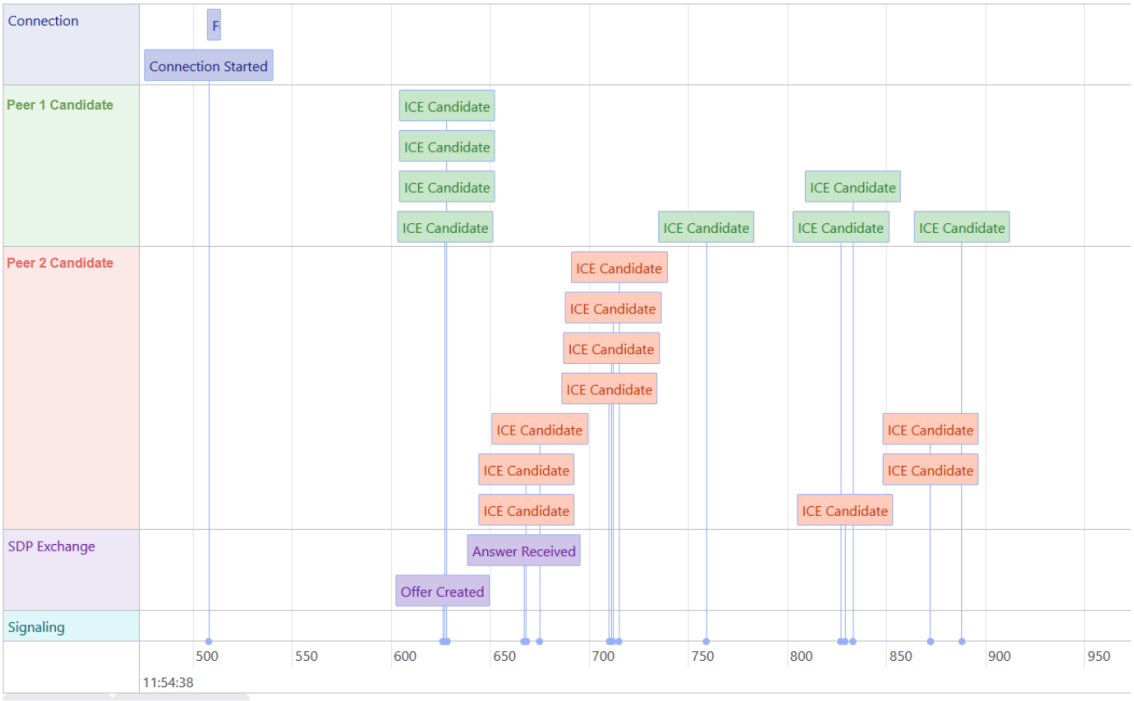


Figure 2.11: Timing of Signaling

2.1.5 WebRTC Data Channel

One of the main components of WebRTC is the data channel, which enables the transmission of any data directly between peers with low latency and high reliability. A data channel's underlying transport layer protocol is the Stream Control Transmission Protocol (SCTP). SCTP is built on top of the network layer, typically over IP, similar to TCP and UDP, shown in 2.12. Data channels also use DTLS to encrypt SCTP data, discussed more in section 2.1.6.

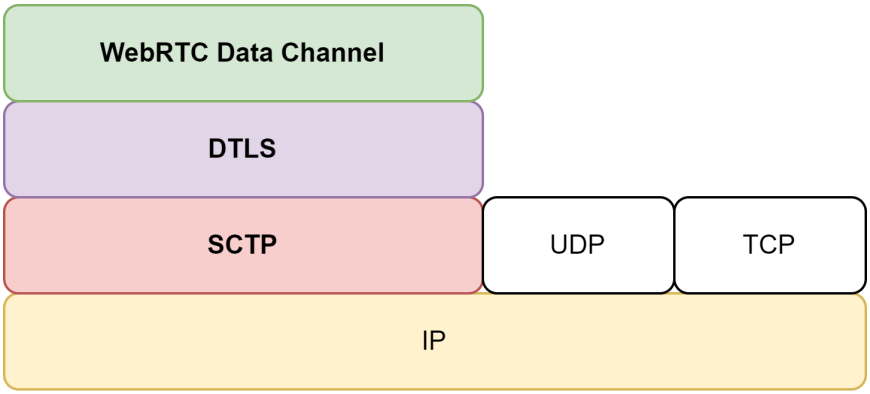


Figure 2.12: SCTP Network Layer

SCTP has a few characteristics:

- **Message-Oriented Communication:** Unlike protocols such as TCP, which treat data as a continuous stream of bytes, SCTP is message-oriented. That means that data is transmitted as discrete messages.
- **Reliable and Unreliable Transmission:** SCTP has multiple operating modes, reliable and unreliable. Reliable operation ensures that no messages are lost and will arrive in order. Unreliable operation means that some messages might be lost after certain time intervals or a number of retries.

PeerProxy, the proposed technique in this thesis, uses the data channel to proxy HTTP messages. Since HTTP needs reliable transport, the data channel is configured to transmit packets in a reliable and ordered manner. More on how Peerproxy serializes HTTP messages over a data channel is discussed in section 3.5.

2.1.6 Data Channel Security with DTLS

To ensure that messages are protected from eavesdropping and tamper resistant, WebRTC uses DTLS or Datagram Transport Layer Security [91]. DTLS can be used on any datagram-based transport, such as SCTP or UDP.

DTLS has a few guarantees:

- **Encryption:** Data sent between peers is confidential and cannot be decoded by anyone but the receiving peer.
- **Authentication:** DTLS uses certificates to ensure the data is from a trusted source.
- **Data Integrity:** Message authentication codes are used to ensure that the data has not been modified in transit.

To achieve this, DTLS uses asymmetric encryption, such as Diffie-Hellman or RSA, to establish symmetric keys. These symmetric keys are then used to encrypt traffic.

Since Peerproxy uses the WebRTC data channel, all application traffic is end-to-end encrypted. Even if the traffic is intercepted, it can't be modified or decoded, ensuring the data's security and privacy.

2.1.7 HTTP

HTTP, or the Hypertext Transfer Protocol, is the primary way web browsers and servers communicate on the World Wide Web. It is an application-layer protocol that can serve different types of data, such as text content, images, videos, and

scripts [79]. HTTP can run on any reliable transport protocol but has mainly been used over TCP [34]. Figure 2.13 illustrates the network layer diagram for HTTP.

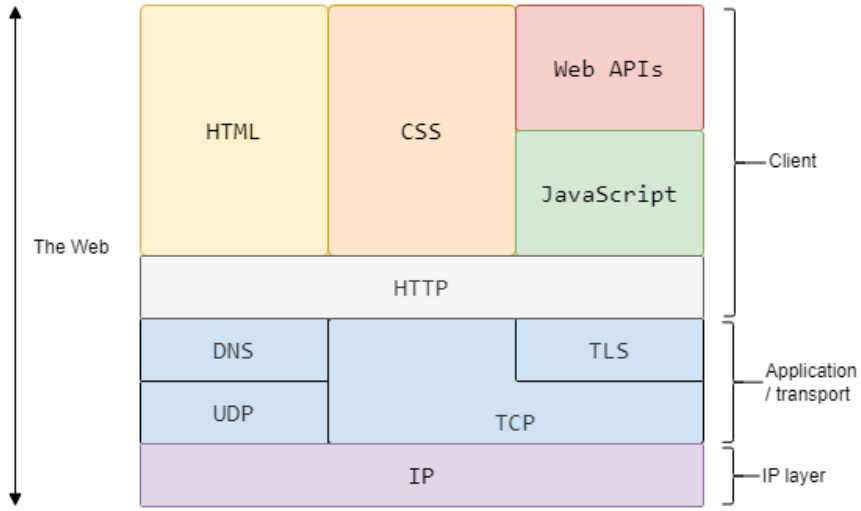


Figure 2.13: HTTP is an Application Layer on top of TCP [33]

HTTP has a request-response model where clients request information from a server, which returns a response.

2.1.8 HTTP Requests

The client initiates requests to ask the server to perform an action, such as returning information or receiving data. An example request is shown in table 2.3.

2.1.8.1 Request-Line

Contains metadata about the request. This contains three components:

Method: Specifies the action to be performed. Common methods include "GET," "POST," "PUT," and "DELETE."

Request-URI: The Uniform Resource Identifier indicates the resource on the server the action is for.

HTTP Version: The version of HTTP such as HTTP/1.1.

2.1.8.2 Request Headers

Provides additional information about the request in the form of key-value pairs. Common values for a request include:

Accept: Indicates the type of data the client is expecting

User-Agent: Identifies the browser or software the client is using.

2.1.8.3 Body (optional)

Used to send data to the server. This is commonly form inputs or file uploads.

Table 2.3: An Example HTTP Request

Request Line	`GET / HTTP/1.1`
Headers	`Accept: text/html` `Accept-Language: en`
Body	

2.1.9 HTTP Responses

The server returns an HTTP response after receiving and interpreting a request message. An example is shown in table 2.3.

2.1.9.1 Status-Line

The status line contains metadata about the response and is slightly different from the request start line. This includes three components:

HTTP Version: The version of HTTP such as HTTP/1.1.

Status-Code: A three-digit code describing the outcome. Common codes include "200" for success or "404" for not found.

Reason-Phrase: A short textual description of the Status-Code

2.1.9.2 Response Headers

Similar to a request, this provides additional information about the response as key-value pairs. Common values for a response include:

Content-Length: the size of the response

Content-Type: the type of data in the response

2.1.9.3 Body (optional)

The body contains the requested resource data, such as an HTML document or image.

Table 2.4: An Example HTTP Response

Request Line	`HTTP/1.1 200 OK`
Headers	`Content-Length: 2026` `Content-Type: text/html`
Body	`<!DOCTYPE HTML />` `...`

One of the main contributions of PeerProxy is proposing an efficient transport of HTTP messages over WebRTC data channels. Since WebRTC data channels are message-based with limited size, PeerProxy splits HTTP messages into chunks and streams them over the data channel. This is covered in more detail in section 3.5.

2.2 Websockets

WebSockets is a communication protocol that enables bidirectional communication between a client and server [70]. This communication happens over a single, long-lived TCP connection. Unlike HTTP, which follows a request-response pattern as described in section 2.1.7, WebSockets allow for the client and server to send messages at any time.

PeerProxy uses WebSockets as the transport for WebRTC signaling (section 2.1.4). WebSockets are an ideal choice for this purpose because they enable peers to receive connection information in real time without the need for polling. They're also lightweight and have nearly full browser compatibility [39].

The actual proxying component of PeerProxy notably lacks support for WebSockets. Any proxied website that tries to make a WebSocket connection to the origin server will fail. Potential workarounds are discussed in section 6.1.2.1.

2.3 Browser Technologies

PeerProxy uses a few browser APIs to load websites seamlessly in the browser client. Below are the main APIs used.

2.3.1 Service Workers

Service workers are a browser API designed to give web applications offline capabilities [37]. They achieve this by acting as a proxy server between the website and the original server.

When a website registers a service worker, the worker is invoked when the website makes network requests, such as fetching images or scripts. The worker can intercept and modify that request before returning it, which enables advanced control over how resources are handled.

The main use case of a service worker is enabling apps to be offline. First, a website has to load normally and cache all the resources required to run offline, as shown in Figure 2.14. Once cached, the client can use the website offline by loading all resources from the cache.

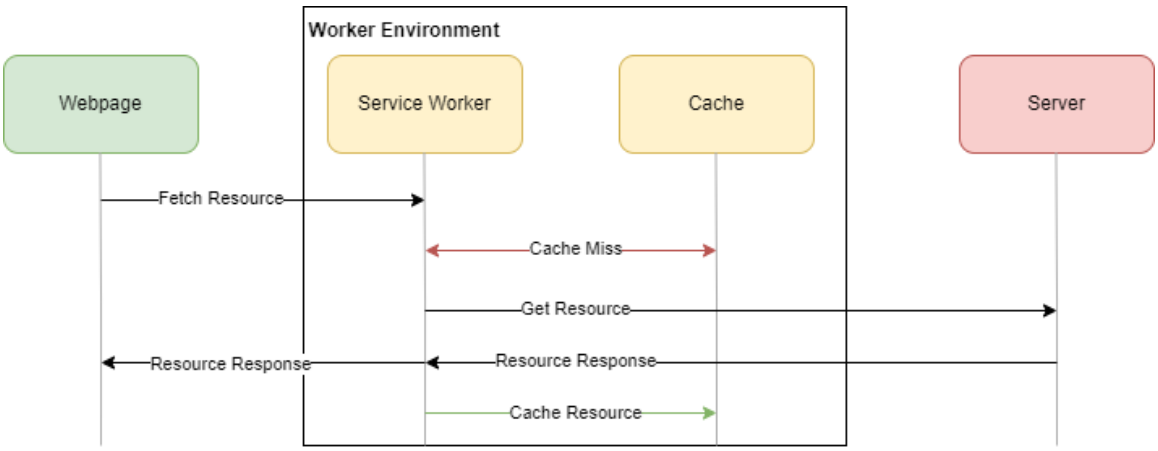


Figure 2.14: Service worker caching resources

PeerProxy only uses service workers for its proxying capabilities, a core part of how PeerProxy loads websites without any modifications. PeerProxy registers a service worker to intercept any request that a proxied request makes. Then, it serializes that request to be sent over WebRTC to the server. When it gets the response over WebRTC, it seamlessly returns the request to the original requester.

Further details on how PeerProxy uses the service worker API will be discussed in section 3.3.2.

2.3.2 Iframes

Iframes (short for "inline frames") are HTML elements that allow a website to create another browsing context within the parent browsing context [30]. Websites usually use them to display content from a third party, such as advertisements, videos, or widgets. A typical example is a website embedding a YouTube video. These browsing contexts are isolated, meaning that the content in the iframe cannot directly affect the parent environment and vice versa.

In PeerProxy, iframes display and isolate the proxied web page instead of loading content from an external site. The sandboxed environment ensures that other components loading the proxied webpage and managing the WebRTC connection do not interfere with the proxied webpage. The sandboxing guarantees that the proxied webpage runs normally without modification.

Loading and running a proxied webpage is discussed further in section 3.6.

2.4 Traditional HTTP Reverse Proxies

Making a local server accessible over the internet can be difficult and usually involves manually configuring port forwarding and firewalls. HTTP reverse proxy services aim to solve that problem by creating a tunnel to a local server and giving the user a publicly accessible URL to access that tunnel. Some popular services include Ngrok [77] and Cloudflare Tunnels [19].

Reverse proxy services usually have a few components. A local proxy server runs on the same computer or network as the server. That process connects directly to proxy servers accessible over the internet with a TCP connection. This infrastructure has a publically accessible URL. When a client accesses that URL, all traffic is sent through the proxy servers and relayed to the local proxy server. These components are shown in figure 2.15.

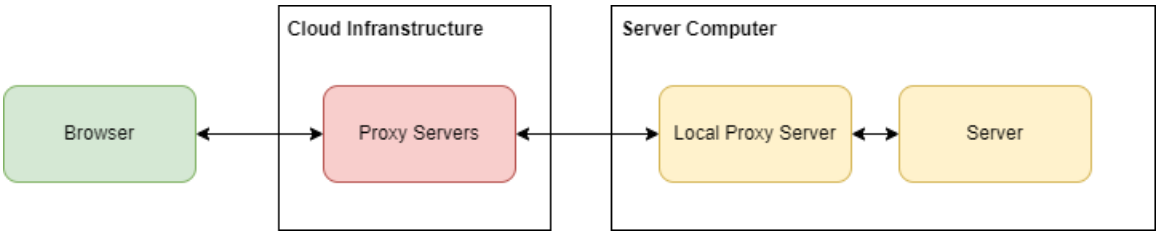


Figure 2.15: Traditional HTTP Proxy

PeerProxy aims to provide the same easy-to-use end-user experience while offering a more efficient and secure transport mechanism. By using WebRTC to directly connect clients and servers, PeerProxy reduces the need for dedicated infrastructure to proxy application traffic. This can enhance privacy, reduce infrastructure costs, and reduce latency in certain cases.

2.5 Related Works

WebRTC is a newer technology and has only recently been standardized in browsers. Since its establishment, work has been done evaluating WebRTC's performance in the real world and on a few interesting applications beyond its original use case of video conferencing. PeerProxy is another application of WebRTC with different use cases than existing applications.

2.5.1 WebRTC Performance Benchmarking

Broad studies have been conducted about WebRTC performance in real-world networks. In a 2018 paper, Moulay and Mancuso examined WebRTC video calls on mobile networks [75]. The authors found that WebRTC calls with sufficient quality are possible in mobile networks in cases where the user is not moving, but quality degrades when users are on the move. They found that increased jitter, or variations in latency, significantly degrade WebRTC performance.

Along with studying WebRTC performance broadly, there has been some research into the performance of WebRTC data channels, a way to exchange arbitrary data with WebRTC (discussed more in section 2.1.5). In a 2015 paper, Eskola and Nurminen investigate WebRTC data channel performance in browser implementations

at that time [41]. The authors evaluated the data channels' performance in different simulated network conditions, focusing on high-throughput applications. The authors concluded that the send and receive windows of the underlying SCTP protocol for the data channel were configured too low. When the authors tried increasing the SCTP window, they got better throughput. However, they ran into issues where the congestion avoidance mechanism built into SCTP heavily cut throughput when there was any kind of packet loss or congestion.

The SCTP window has been raised in recent versions of Firefox and Chromium since 2015, but congestion control is still a potential limiting factor for high throughput applications. This thesis does an evaluation of current performance in browsers in 2024 5.1.3.

2.5.2 Censorship Resistant Proxying with WebRTC

One innovative application of WebRTC is as a censorship-resistant proxy. In a 2020 paper, Barradas et al. introduced Protozoa, a tool that utilizes WebRTC to tunnel IP traffic through WebRTC video streams [9]. Since traffic replicates normal video streams, it's harder to detect and block. Protozoa works by hooking into the WebRTC stack of an unmodified browser. A user would then start a video call using a 3rd party video conferencing platform to someone in an uncensored area with a special proxy attached to their browser. The authors used Protozoa to effectively evade censorship in countries such as China, Russia, and India. They achieved higher bandwidth and better detection than similar methods.

This technique was further refined with the tool Stegozoa, proposed by Figueira et al in 2022, which adds video stenography, where proxied data is embedded in video frames. This adds resilience against more advanced machine-learning-based

censorship tools. The authors were able to maintain reasonable throughput while achieving state-of-the-art censorship resistance [45].

These techniques have been used in the real world with the tool Snowflake [15]. Snowflake is a Tor plugin designed to help users maintain internet access in countries with state-wide censorship. It achieves this by connecting users from censored regions to thousands of lightweight, volunteer-run proxies using WebRTC. In a paper evaluating Snowflake in 2024, Bocovich et al. found that the plugin has proven effective in countries such as Iran and Russia. This plugin is widely used, with the network having an average of 35,000 concurrent users and a daily transfer of 29TB.

2.5.3 Novelty of this Work

This section emphasizes the novelty of this thesis with respect to previous work done. Similar to Protozoa and Snowflake, PeerProxy is an application of WebRTC that goes beyond video conferencing. PeerProxy aims to be an alternative to commercial reverse proxies like Ngrok discussed in section 2.4. One of the main contributions of this thesis is a functional prototype of PeerProxy that can serve websites from local networks anywhere. This includes a custom web-based client, signaling server, and local proxy server.

Additionally, this thesis also has a performance comparison between PeerProxy and a commercial reverse proxy Ngrok, evaluating metrics such as throughput, latency, and resource utilization. These tests were run using a custom-built testing application between two Amazon EC2 (Elastic Compute Cloud) instances.

This thesis also provides an update to the findings of Eskola and Nurminen [41] in section 5.1.3. Since that paper was published in 2015, the SCTP window has been raised from 128 KB to 2MB in the Chrome browser [102]. This thesis runs similar tests to evaluate the current performance of WebRTC data channels in modern browsers.

This work demonstrates the feasibility and practicality of using WebRTC to access and proxy local web services. In doing so, it offers a comprehensive performance evaluation that highlights the benefits and limitations of this approach. Then, this work expands to explore the current limitations of performance for high throughput WebRTC applications in modern browsers.

Chapter 3

IMPLEMENTATION

3.1 Peer Proxy Overview

PeerProxy enables users to easily proxy locally served web servers over the internet. This means that users can access web applications hosted on their local machines from any device, anywhere. As opposed to regular HTTP reverse proxy services such as the ones described in section 2.4, PeerProxy transports all application data over WebRTC.

All code for PeerProxy, including the signaling server proxy server, browser client, and benchmarking applications is available at: https://github.com/Nathanllee1/PeerProxy.

3.2 Requirements for the PeerProxy Design

PeerProxy aims to provide the convenience of using a reverse proxy service while increasing privacy and reducing infrastructure with minimal trade-offs. Several key requirements were established for PeerProxy's design to meet this objective. Not only are these requirements relevant to the design, but they also provide a framework for evaluating PeerProxy, discussed in section 4.

The following are requirements that guided the design of PeerProxy.

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [16]

1. The browser client MUST be compatible with unmodified modern browsers and not require installation beyond loading a web page.

2. The overall system MUST proxy any HTTP request from the browser.

3. The proxy SHOULD be able to serve websites without modification. The websites retain their full functionality and appearance, ensuring they look and behave as if the website was served directly from their original servers.

4. The latency and speed of web requests SHOULD be optimized to ensure a user experience similar to a commercial proxy.

5. The server proxy SHOULD be resource efficient and handle concurrent requests from multiple clients.

6. The browser client SHOULD add minimal overhead to loading and rendering webpages.

3.3 PeerProxy Components

The local server is the web application, such as a Node.js app, a Python Flask service, or any other HTTP-based site that the user wants to make accessible from anywhere. To accomplish that, PeerProxy has three primary components.

1. **Local Proxy Server:** an executable that runs on the user's machine that receives WebRTC connections and forwards requests to the user's local server. This is written in Go [84] and primarily uses the library Pion [85] for WebRTC.

2. **Browser Client:** a web application served publicly with a CDN that connects to the local proxy server and loads the website, just like if it was loaded normally.

3. **Signaling Server:** coordinates the exchange of connection details between the local proxy server and the local proxy server to form a WebRTC connection.

A sample use case is if a user has a web application running on their local computer (the local server). Then, the user launches the PeerProxy local proxy server on the same machine. Under the hood, this local proxy server receives WebRTC connections from inside and outside the local network and forwards HTTP messages to the local server. Data sent over a WebRTC data channel is sent in packets, so the local proxy server deserializes requests from the browser client and serializes response from the local server. The local proxy server connects to the signaling server and is provided a URL to the PeerProxy browser client with an identifier like "id.peerproxy.dev".

A remote user who wants to access the web application simply visits the URL for the local proxy server. The browser client uses that connection identifier to connect to the signaling server to send and receive connection information from the local proxy server. Once a WebRTC connection is formed, the website and all application data are sent through that secure connection. The individual components of the browser client are discussed in section 3.3.1.

These components are shown in figure 3.1.

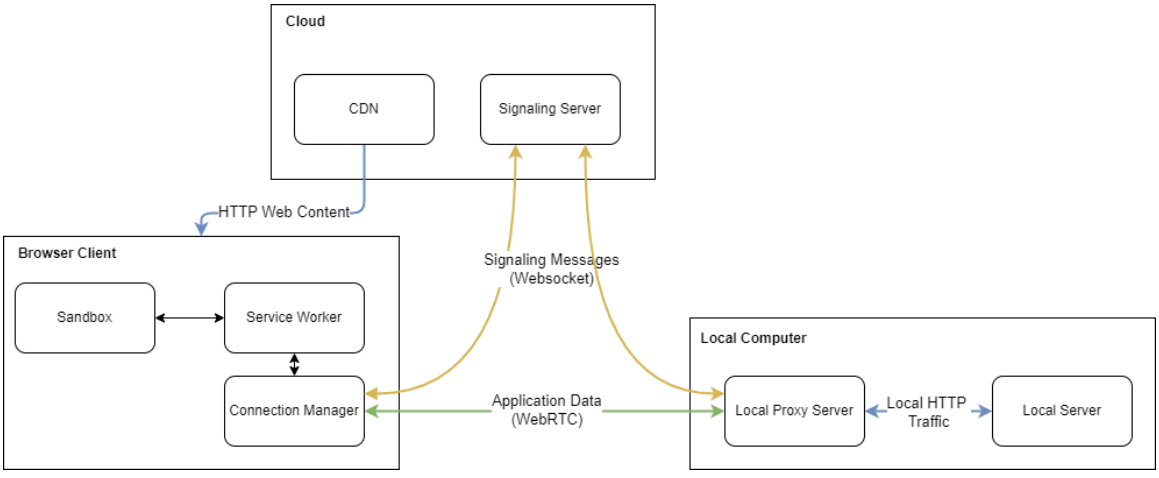


Figure 3.1: High-Level Overview of PeerProxy Components

3.3.1 Browser Client Components

The browser client loads and displays the webpage. The client is a normal web app hosted on a CDN (Content Delivery Network) and has three components: the connection manager, sandbox, and service worker.

Connection Manager: The connection manager manages the WebRTC connection to the local proxy server. It handles all the signaling with the signaling server to form a WebRTC connection. The connection manager has interfaces that let other parts of the browser client send and receive packets from the WebRTC connection. The connection manager is located in the main context of the browser client since the service worker environment does not support WebRTC.

Sandbox: Once a connection is established between the connection manager and the local proxy server, the website and its resources are loaded and rendered in the sandbox. More about how the sandbox renders a page is discussed in section 3.6. A service worker is registered for the sandbox and intercepts any network requests made within the sandbox. Those requests get proxied to the local proxy server. The sandbox is implemented with an iframe to isolate the proxied page in its own environment. Iframes are discussed in section 2.3.2.

Service Worker: This is the mechanism used to intercept any request that a webpage in the sandbox makes. The service worker runs in a separate context and is bound to the iframe. It uses the "fetch" event handler, which gets called every time the proxied page makes a request. This event handler mechanism is described further in section 2.3.1. The fetch event is serialized into packets and sent to the connection manager to go to the local proxy server. Once the service worker gets the response, the original caller is given a response.

3.3.2 Request Flow

First, the browser client establishes a connection to the local proxy server with WebRTC. This is discussed step by step in section 3.4. The request goes through the components above in the following order, also shown in figure 3.2.

1. First, the sandboxed webpage within an iframe makes a request. This could be a request for the initial page or an external asset like an image or script.
2. The service worker intercepts that request and stops it from going to its original destination.
3. The service worker serializes the HTTP request into packets. This is further discussed in section 3.5.1. As the packets are read, they are sent back to the connection manager.
4. The connection manager sends the packets over the WebRTC data channel to the local proxy server.
5. The local proxy server receives the packets and deserializes them into an HTTP request to the local server. These packets are ordered and sent on the fly to allow for streaming requests. This is discussed in section 3.5.2
6. The local server receives the HTTP request and sends back an HTTP response.
7. The local proxy server serializes the response into packets and streams it over the WebRTC data channel.
8. Then, the connection manager receives the packets and forwards them to the service worker.
9. The service worker deserializes the packets, and they are streamed back to the original requester.

10. The original requester seamlessly gets the data exactly as if it was requested normally.

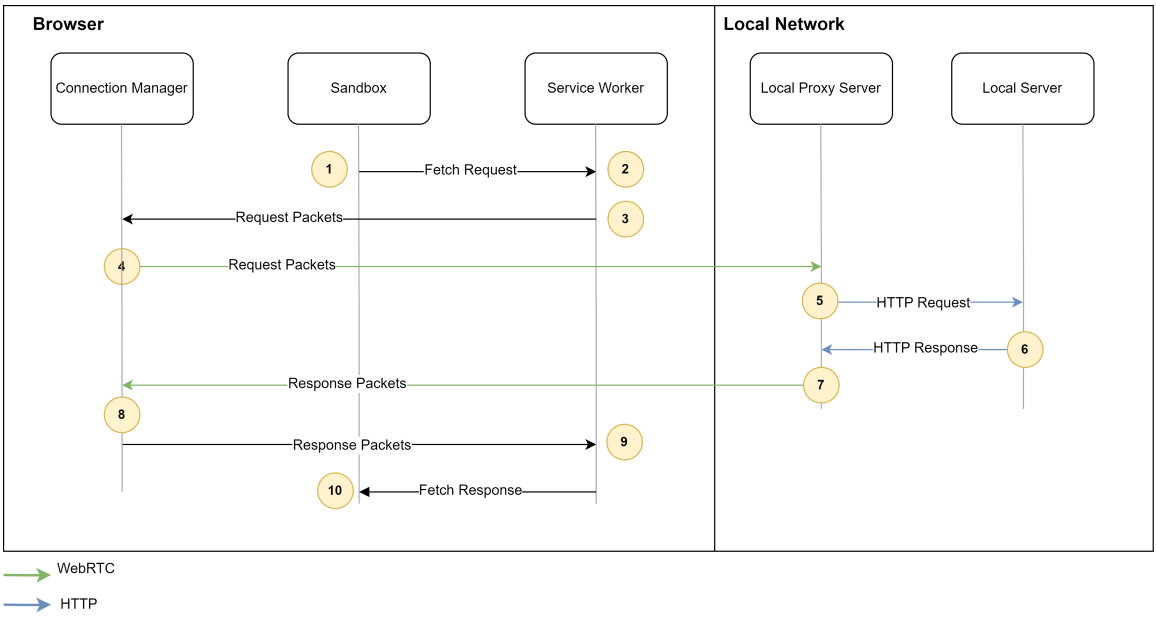


Figure 3.2: How a request is made and proxied

3.4 Establishing a Connection

PeerProxy must first form a WebRTC connection between the browser client and the local proxy server. As described in section 2.1.4, to establish a WebRTC connection, there needs to be a signaling server through which both peers can exchange information. There isn't a standardized solution defined in the specification, which has led to a significant fragmentation of methods, causing most projects to have their own signaling methods. Along with facilitating signaling, PeerProxy's signaling process also manages server registration, which necessitated the development of a custom signaling implementation.

PeerProxy and most WebRTC signaling implementations use WebSockets (described in section 2.2) since they allow for efficient bidirectional communication and are supported in most environments.

The signaling implementation uses a simple JSON packet format. A "mtype" field specifies the message type, such as "offer," "answer", etc. The recipient field is the peer's ID, which the server uses to route the packet to the right user. The payload field stores the actual signaling information.

```
{
   mtype: ‘‘idRequest’’ | ‘‘offer’’ | ‘‘answer’’ | ‘‘candidate’’,
   recipient: <id>
   payload: {
        ...
   }
}
```

Signaling goes through the following steps and is also shown in figure 3.3

1. The local proxy server establishes a WebSocket connection to the signaling server and sends an "idRequest" to the signaling server. The local proxy server can register as a specific ID or be assigned one. The signaling server sends the assigned ID back to the local proxy server.

2. The signaling server stores that connection under an id.

3. When a browser client wants to connect to a given ID, it'll establish a WebSocket connection to the signaling server.

4. The browser sends an "offer" to the signaling server and is forwarded to the local proxy server

5. On receiving the "offer," the local proxy server creates an "answer." That gets sent to the signaling server and forwarded to the browser client

6. As ICE candidates get discovered, they get sent to each other. Each client tests pairs of candidates until a working connection is found.

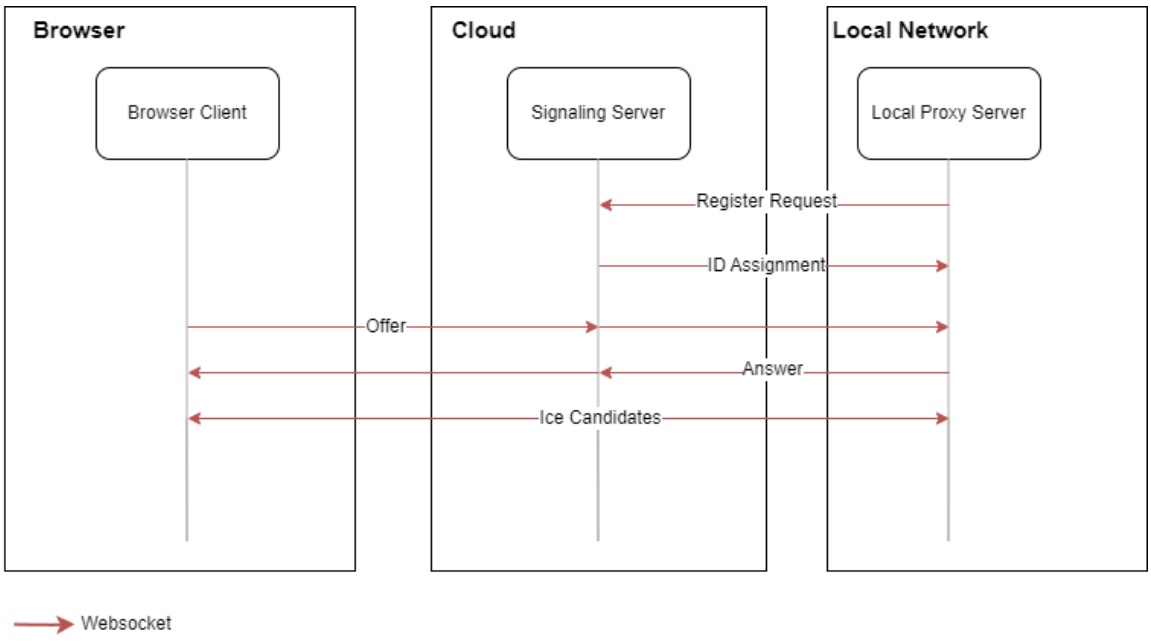


Figure 3.3: PeerProxy Signaling Server

For a proof of concept signaling server, PeerProxy uses a single NodeJS instance in a docker container hosted on fly.io [46] with 512 MB of RAM. This setup is limited but more than sufficient for a proof of concept. This approach is the minimum for signaling and was implemented for PeerProxy only as a proof of concept. This thesis focuses on the proxying of HTTP requests over WebRTC, not the initial establishment of WebRTC.

3.5 Packet Protocol

One of PeerProxy's main contributions is the efficient transport of HTTP over a WebRTC data channel. The data channel is designed to exchange arbitrary data between peers. Since it uses SCTP under the hood, it is message-oriented, meaning that data is treated as discrete messages.

Due to browser limitations and incompatibilities, MDN recommends limiting packet sizes to 16kb [76]. There is also ongoing work to implement a stream scheduler to send arbitrary-size messages without blocking the channel, but it is still in the IETF draft stage as RFC 8260 [98].

Since many web resources are larger than 16kb, implementing a chunking system to split larger responses into chunks is the natural solution. PeerProxy uses a custom packet protocol to transmit data of arbitrary size over a data channel. The packets consist of a header section with metadata and a payload section containing the actual data, as shown in figure 3.4.

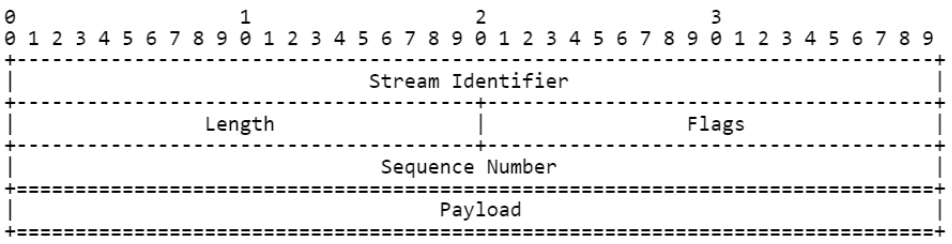


Figure 3.4: PeerProxy Packet Protocol

The header packet contains several fields:

Stream Identifier (unsigned 32-bit integer): This identifies which HTTP stream this packet is part of. Since it's an unsigned 32-bit integer, PeerProxy can handle up to 2^{32} requests on a single connection until the identifier is reset back to 0.

Length (unsigned 16-bit integer): The length of the data in the payload.

Sequence Number (unsigned 32-bit integer): Indicates the position of the current frame within the overall sequence of frames.

Payload (variable-length sequence from 0 bytes up to 16kb - 12 bytes): This contains the actual data stored in the packet

Flags (16 bits): Each bit represents boolean flags describing the packet.

The flags below in table 3.1 are used in the current PeerProxy implementation, but more can be added.

Table 3.1: Flags

0 - 13	14	15	16
Unused	Heartbeat	Message Type	Final Message

Heartbeat: Whether the packet is a heartbeat. It keeps the connection alive, and the packet is discarded upon receiving.

Message Type: If selected, the packet is type *BODY*, otherwise it is type *HEADERS*.

A packet marked with *BODY* contains a portion of the HTTP body in the payload.

A packet marked with *HEADERS* contains the request's headers in the payload. In the future, header data will be compressed using HPACK [82], a header compression format used in HTTP2. There is a lack of client-side javascript HPACK libraries, so headers are stored as ASCII JSON for the initial implementation.

Final Message: Boolean flag signaling whether the current packet is the last message in a series of packets.

3.5.1 Packet Serialization

Packet serialization turns an HTTP response or request into packets to be sent over a data channel. The HTTP message objects in the local proxy server and browser client contain a header object and a body that can be streamed. The serialization routine first turns the headers into a packet and sends it. Then, it consumes the body stream and outputs a new packet every 16kb read. These packets are sent on the fly to maximize performance. This is shown in figure 3.5

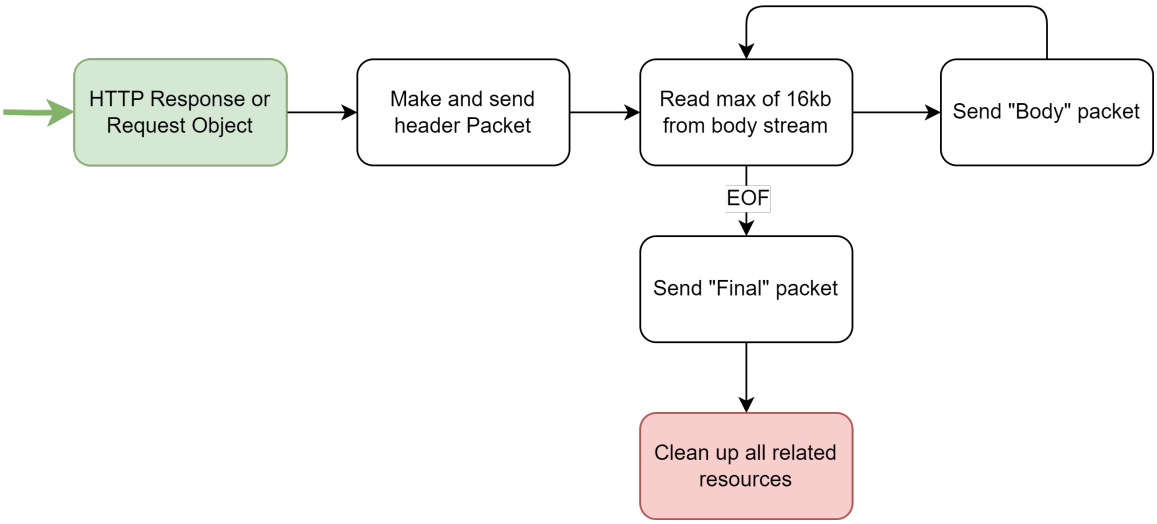


Figure 3.5: Packet Serialization

Packet serialization is implemented in 2 places, also shown in figure 3.6:

Browser client: The browser client intercepts any request the web page makes as a "fetch" event. This "fetch" event is turned into a series of packets that get sent to the local proxy server.

Local Proxy Server: The local proxy server serializes packets once it gets the response from the local server. The response headers are sent immediately, and the body is streamed into packets sent as they are consumed.

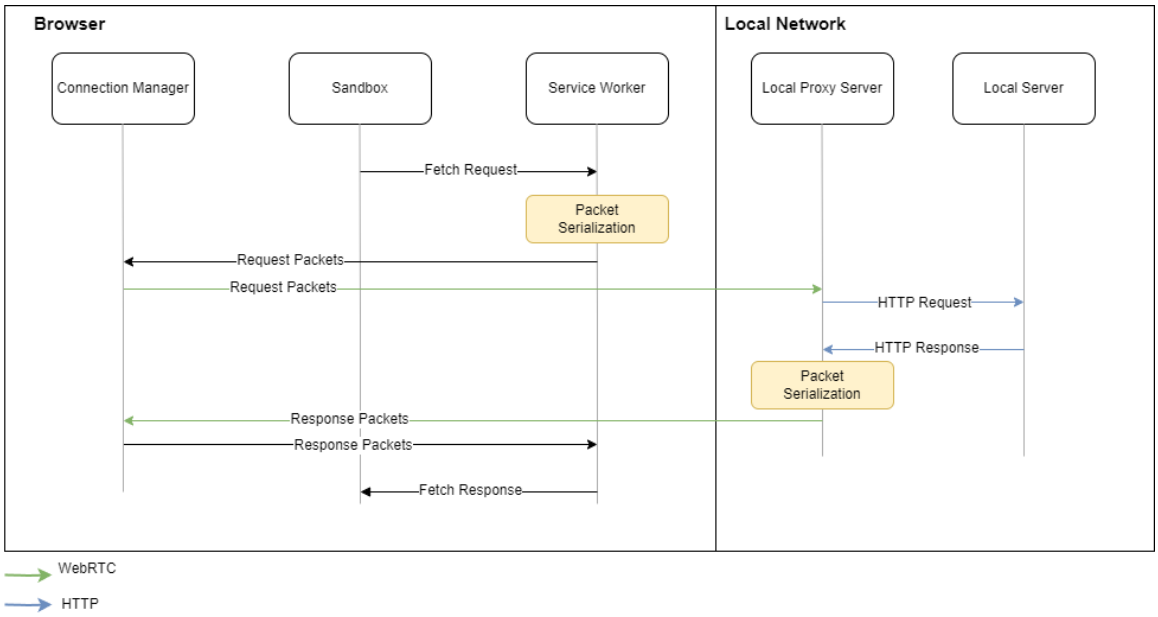


Figure 3.6: Packet Serialization Locations (shown in yellow)

3.5.2 Packet Deserialization

Packet deserialization turns packets received in a data channel into an HTTP request or response. Each client has a callback that receives a packet through the data channel. The packets are parsed and sent to the packet ordering routine. The packet ordering routine outputs a stream of bits replicating an HTTP request or response. Figure 3.7 shows the packet deserialization approach in detail.

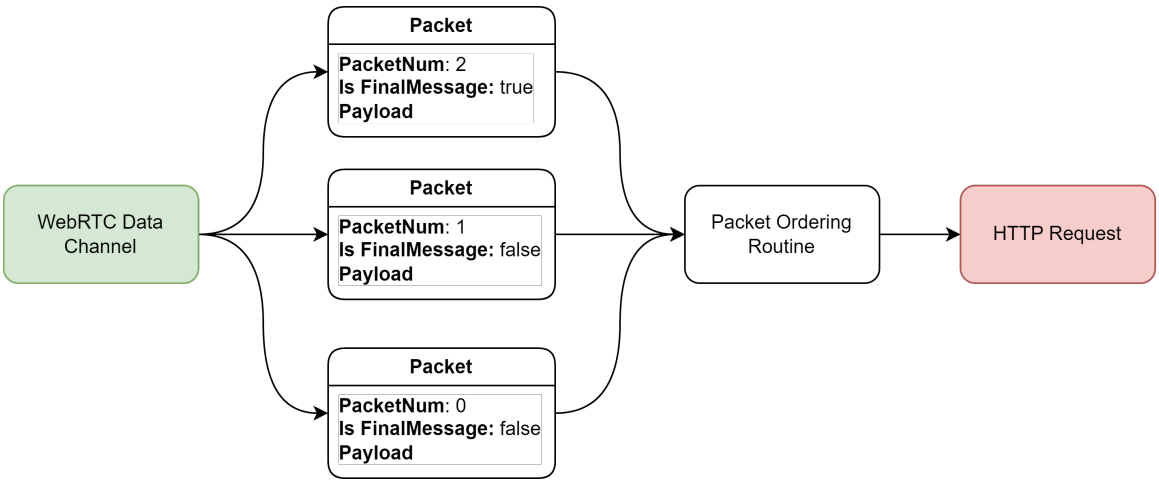


Figure 3.7: Deserialization

This reassembly process ensures that the data is reconstructed accurately, preserving the integrity of the original HTTP message. Since the output is a streaming body, this replicates how HTTP handles continuous data transmission. Streaming data also ensures that PeerProxy can efficiently handle large amounts of data with minimal resources and low latency.

Packet deserialization is implemented in 2 places and is shown in figure 3.8:

Local Proxy Server: Every packet received in the data channel gets put into the streaming body of the HTTP request.

Browser client: Every packet received in the data channel gets put into the streaming body of the HTTP response.

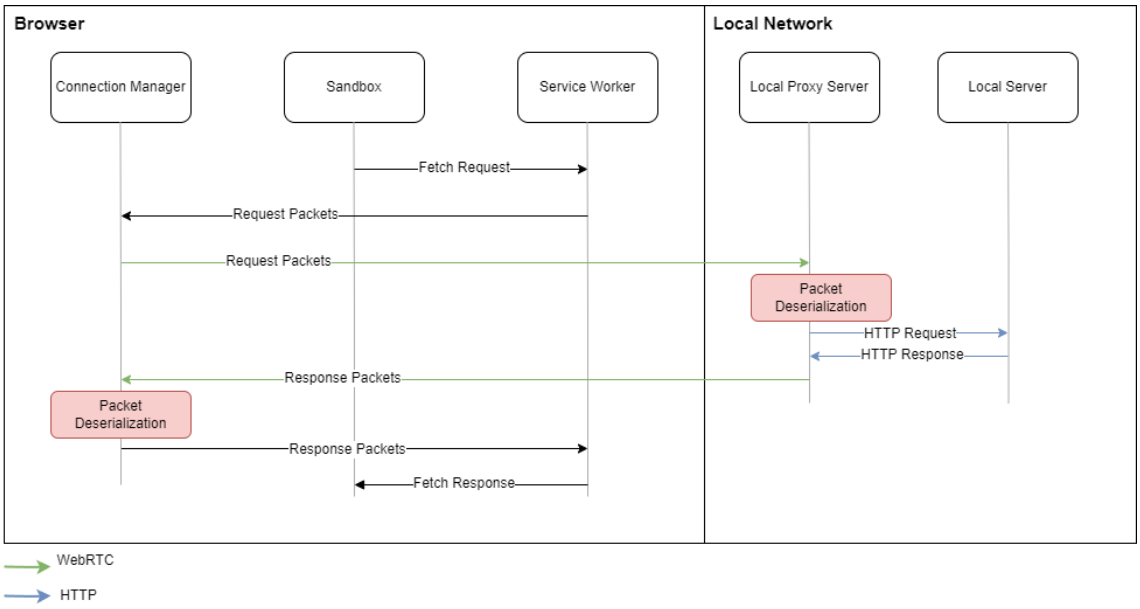


Figure 3.8: Packet Deserialization Locations (shown in red)

3.5.2.1 Packet Ordering

The WebRTC data channel has built-in packet ordering, ensuring packets are received in the correct sequence. However, packets are deserialized concurrently in the local proxy server and browser client to maximize performance.

Handling packets concurrently and directly outputting them into the output stream can introduce race conditions. For example, the local proxy server spawns a new goroutine, an abstraction over threads, on every packet received. As shown in figure 3.9, suppose this made two threads, each processing a different packet in parallel. They would eventually add the packet into the output HTTP stream, but there's no guarantee that the two threads will finish in the order they were called. The threads may put the data into the output stream in the wrong order, leading to data integrity issues.

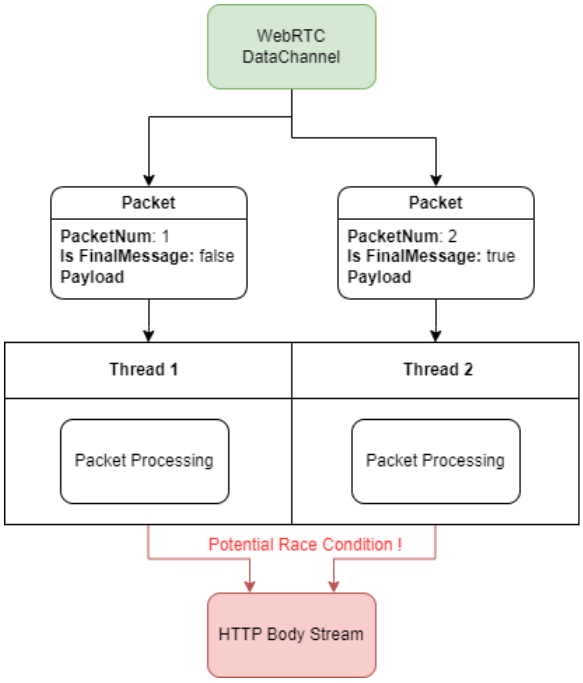


Figure 3.9: Race Condition Example

To deserialize packets in a thread-safe and concurrent manner, PeerProxy implements a packet-ordering routine shown in figure 3.10. This happens in the "Packet Ordering Routine" shown in figure 3.7.

This routine is called every time a new packet arrives. A thread-safe object keeps track of the current packet number, the last packet, and an out-of-order packet map.

If the packet received is the last, the last packet field is set to that packet number. The stream doesn't end since there might still be packets before the last packet that haven't been sent through. But if all packets have been found, the HTTP stream is closed.

Next, if the packet is next in order, it is sent through the stream, and the current packet number is incremented. But if it isn't next in order, it is added to the out-of-order packets map.

Finally, the routine checks if any packets in the out-of-order packet map are the next packets and adds them to the stream. It keeps looping until there are no more sequential packets.

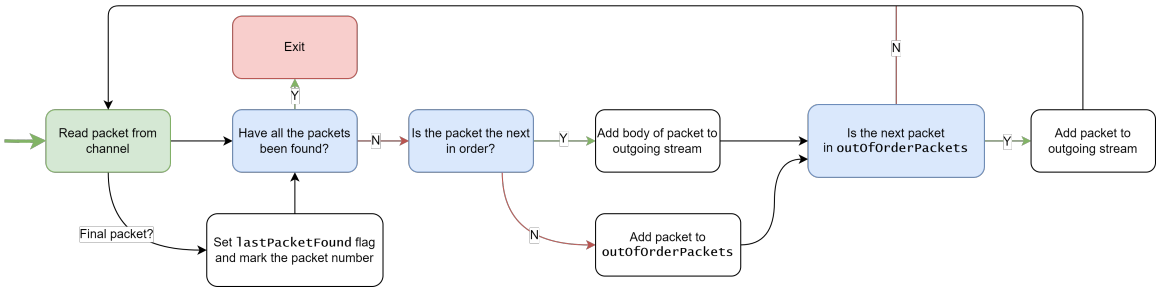


Figure 3.10: Packet Ordering Algorithm

This algorithm ensures that all packets get deserialized accurately with high performance.

3.6 DOM (Document Object Model) Construction

Once a secure connection is established, there is a reliable way to send arbitrary amounts of data, and the service worker is set up to intercept requests, PeerProxy can load the website in the sandbox.

As described in section 2.3.2, PeerProxy uses iframes to isolate the proxied webpage. The iframes ensure that the proxied webpage is isolated from the rest of the browser client and runs as expected. After some research, the `document.write` [32] API is the most reliable way to render a website from text. The webpage is loaded into the iframe using the following steps:

1. The sandbox does a manual GET request for the root page from the current page URL.

2. The service worker intercepts this request and sends the request to the server.

3. When the sandbox gets the response text, the sandbox uses the `document.write` API to write the HTML into the iframe.

4. The browser parses the text and loads linked resources such as JavaScript and CSS files.

3.7 Client Side Routing vs Full Page Refreshes

Establishing a WebRTC connection is much slower than a standard TCP connection. More on connection times will be explored in the results section 5.4, but a typical connection takes 0.5-3 seconds, depending on the network setup.

This connection time adds 0.5-3 seconds on every single page load, including page navigations, since the entire browser client is refreshed and has to make a new connection. This increase in load time negatively impacts user experience and efficiency from the initial load, and especially on page navigations. Page navigations with a full refresh are shown in diagram 3.11.

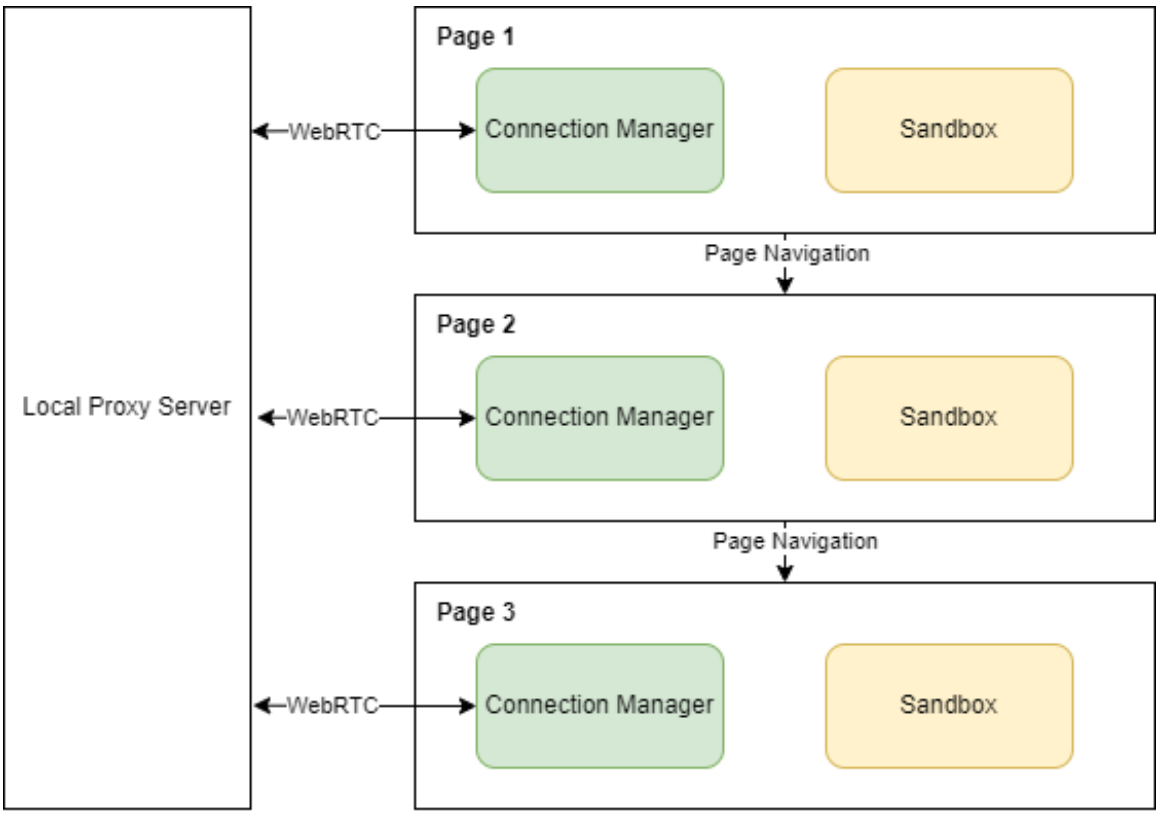


Figure 3.11: Original Page Navigation

One potential solution to speed up navigation is implementing a client-side routing solution inspired by SPAs (single-page applications). Instead of reloading the entire browser client on each navigation, only the sandbox would be updated with the new page. That way, the connection manager would remain connected to the local proxy server and would not have to do a long reconnect. Reusing the connection is illustrated in figure 3.12.

To reuse the connection, the sandbox would intercept all page navigations and prevent the overall page from navigating. Then, the new page would be fetched normally as described in section 3.6. This brings connection time down to 0 since the WebRTC connection is reused.

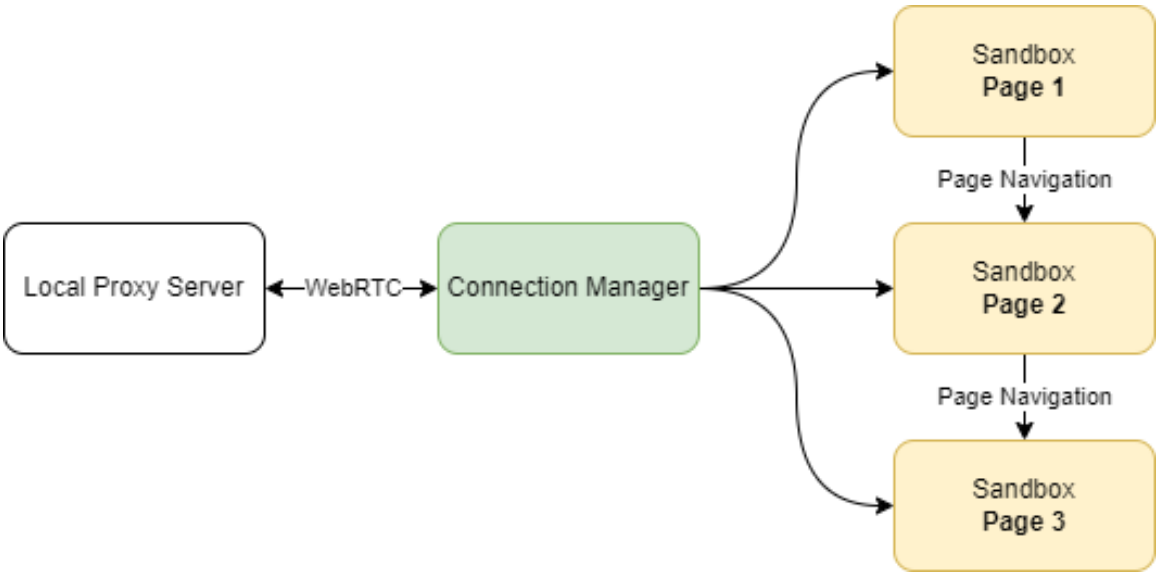


Figure 3.12: Single Page App Routing Style Approach

Ideally, intercepting page navigations would be handled by the Navigation API. The Navigation API has an easy interface that intercepts any page navigation. However, during the development of PeerProxy, this API was still considered experimental by MDN and not supported by Safari or Firefox [63].

Instead, PeerProxy has to handle several types of page navigations manually and use the History API. This is a less streamlined approach, but similar approaches are used in several other client-side routing libraries. The first type of navigation is intercepting any clicks on link elements. To do this, PeerProxy registers a listener for any clicks on the webpage. Then, it goes up the DOM tree and checks if any parent elements are link elements or 'A'. If there is a link element, the page's history is updated using the History API. Another type of navigation is if the user clicks the browser's "back" or "forward" buttons. This is a simple listener on the event 'popstate' that calls the History API with the new page. Once any navigation events are triggered, the sandbox removes the old iframe and loads a new one with the new page as described in section 3.6.

3.8 Security

The main focus of this implementation of PeerProxy was not specifically on security, but it inherently incorporates some security features. One potential attack vector in networking situations is a Man-in-the-middle attack (MITM). MITM attacks occur when an attacker sits between two parties and intercepts communication [47]. The attacker could potentially read or even modify communication. In PeerProxy, the risk of an MITM attack arises in a few places, notably the WebRTC data channel.

PeerProxy uses the WebRTC data channel to transfer all data between the local proxy server and the browser client. This channel is particularly sensitive to attacks, as it handles the transmission of potentially sensitive user data. As discussed in section 2.1.6, the WebRTC data channel uses DTLS (Datagram Transport Layer Security) [91]. DTLS ensures all messages are encrypted end-to-end, preventing unauthorized access by intermediaries. Additionally, DTLS also ensures data has not been modified in transit. Even if a relay server is compromised, an attacker wouldn't be able to view or tamper with data.

End-to-end encryption is not typically supported by commercial third-party proxies [50]. For example, Ngrok doesn't have end-to-end encryption for HTTP tunnels [109]. Even though their tunnels use HTTPS, data is still decrypted to be routed in the proxy servers.

Although efforts have been made to incorporate fundamental security measures into PeerProxy, security was not the primary objective of this implementation. Consequently, there may exist vulnerabilities or unforeseen attack vectors not fully addressed by the current design. While features such as DTLS encryption within We-

bRTC channels significantly mitigate certain security threats, complete protection against all potential security risks cannot be guaranteed.

Chapter 4

EXPERIMENTAL SETUP

The goal of evaluation is to evaluate whether PeerProxy meets the requirements specified in section 3.2. The following setup and environment were created to determine what the tradeoffs are between a commercial reverse proxy like Ngrok [77] and PeerProxy.

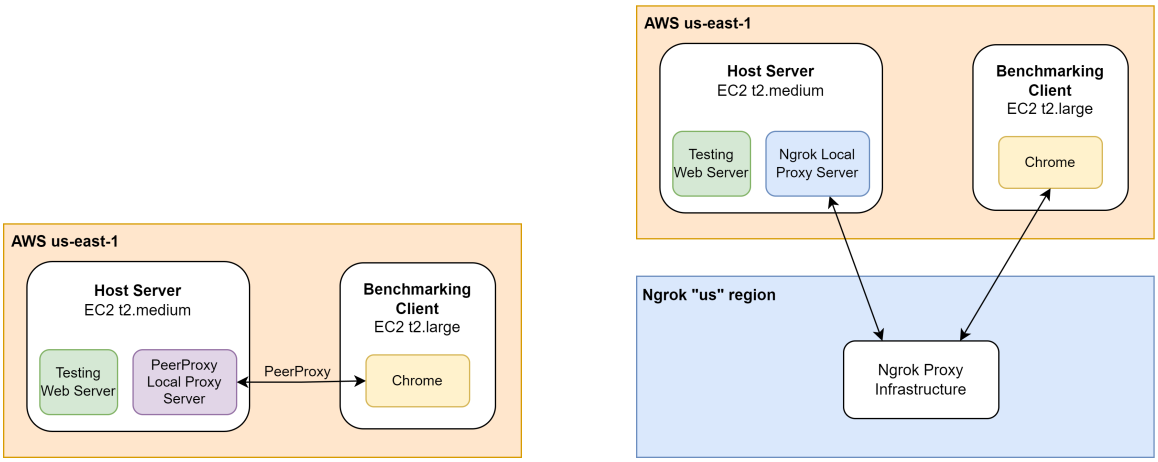
4.1 Experimental Environment

The experimental environment to benchmark PeerProxy was conducted using virtual machines hosted on Amazon Web Services (AWS) [6], specifically using Elastic Compute Cloud (EC2) instances. AWS EC2 is a cloud computing service that provides virtual machines at different compute capacities [4].

The environment consisted of two EC2 instances in the AWS region us-east-1. The first instance was a t2.medium, which served as the host server, running a testing web server (details in section 4.3) and the Ngrok and PeerProxy clients. The second instance was a t2.large, which acted as the benchmarking client and connected to the host server by launching a headless Chrome instance (further discussed in section 4.2). The setup for these instances are shown in figure 4.1.

Table 4.1: EC2 Instance Types Used [5]

Instance	vCPU	Mem (GiB)
t2.medium	2	4
t2.large	2	8



(a) PeerProxy Evaluation Setup *(b) Ngrok Evaluation Setup*

Figure 4.1: PeerProxy and Ngrok Evaluation Setups

4.1.1 Comparing Against a Non-Optimal Proxy

Ngrok has proxy servers in multiple regions to achieve the lowest latency. For testing purposes, two configurations were used: one where the proxy server was correctly configured to the nearest region for optimal performance and another where the proxy server was intentionally configured in Europe to simulate higher-latency conditions.

One use case for PeerProxy is that it eliminates the need to set up a global proxy infrastructure. For example, an IoT company with most of its customers in Europe might deploy its proxying infrastructure in Europe to optimize for its customer base.

However, if a customer in the US wants to interact with a device through the infrastructure, the request would first go to the proxy servers in Europe and then travel back to the device or service. This would significantly increase RTT (round trip time), leading to increased latency and decreased throughput.

To test this, the experimental setup with the two EC2 instances in "us-east-1" was the same, but the Ngrok proxy was configured to use the "EU" region. Requests went

from the Benchmarking Client to the Ngrok Proxy Infrastructure in the "EU" region back to the Host Server.

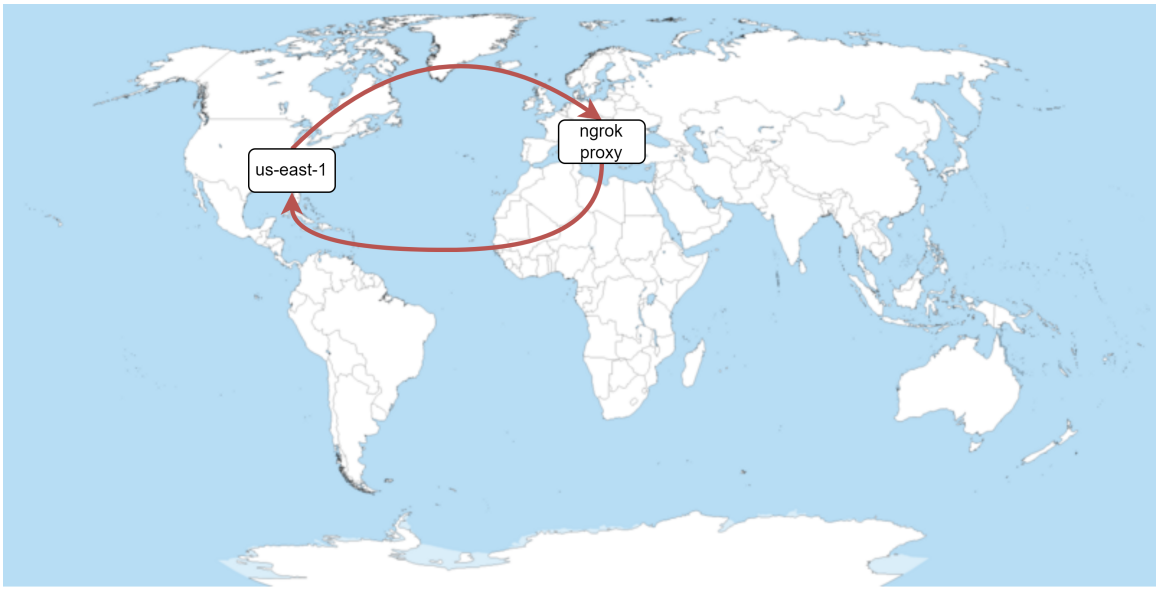


Figure 4.2: Non-Optimal Proxy Setup

4.2 Benchmarking Client

The benchmarking client server was used to connect to the host server. Since PeerProxy only supports a web browser as a client, the benchmarking tests ran tests with Google Chrome [62]. To ensure consistent results, the tool Puppeteer [24] was used to programmatically create and control a headless Chrome instance.

4.3 Testing Web Server Setup

The testing web server is a simple web server written using the Node.js framework Express [42]. It serves a static directory that is a benchmarking site. This benchmarking site uses Javascript to run a series of tests by making requests to a REST

API and recording information about the round trip time of those calls. The basic architecture is shown in figure 4.3.

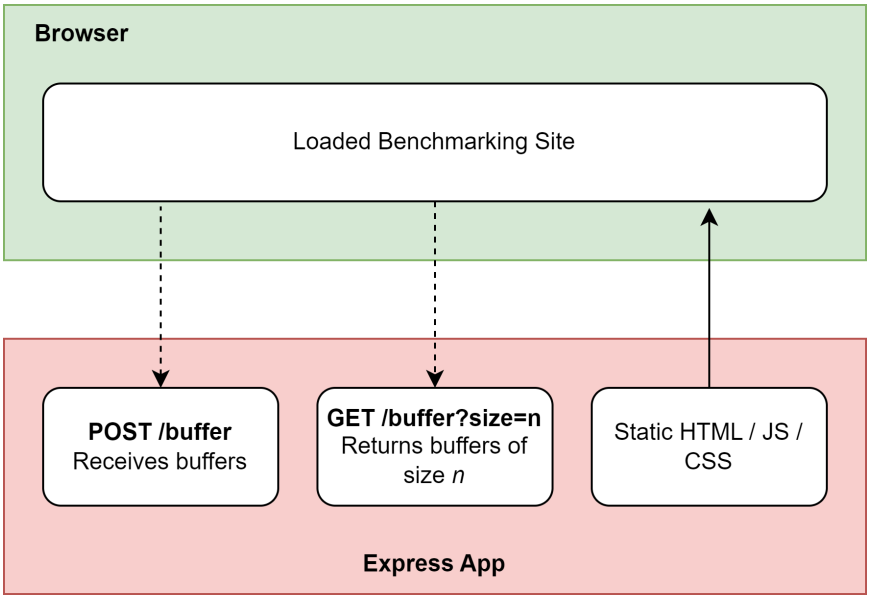


Figure 4.3: Host Server

To ensure that the testing web server was not a performance bottleneck, it was run using PM2, a process manager for Node.js applications. PM2 optimizes performance by forking the server across multiple CPU cores and providing built-in load balancing to efficiently distribute incoming requests [86].

Chapter 5

RESULTS

The following are the performance testing results of the experimental setup discussed in section 4. These tests evaluate the latency, throughput, and resource utilization of a web application served through PeerProxy versus Ngrok. In section 5.5, there's a discussion about the environments in which PeerProxy can run.

5.1 Browser Based Tests

The browser-based tests were run using the benchmarking site as discussed in section 4.2. The tests were a series of requests to the "/buffer" API endpoint, which uploads and downloads data from the server. The benchmarking site records the amount of data transferred and how long each request took. Once the tests are completed, results are compiled into a CSV and downloaded to the benchmarking client.

5.1.1 Throughput

The throughput tests are designed to measure the transfer rate between the browser client and the local proxy server. The tests were performed sequentially, with each new request waiting until the previous one completes. The payload sizes ranged from 0 MB to 5 MB, with increments of 50 KB between each size. Each payload size was downloaded five times from the local proxy server to the browser to minimize outliers and average out results.

After data collection, linear regression analysis was applied to model the relationship between data size and loading time. The relationship was observed to be linear, so overall throughput is the slope of the line, and base latency is the y-intercept.

The following tests evaluated the amount of time it took to upload and download various sizes of data from the server. The raw data for download is shown in figure 5.1, and upload is shown in figure 5.2.

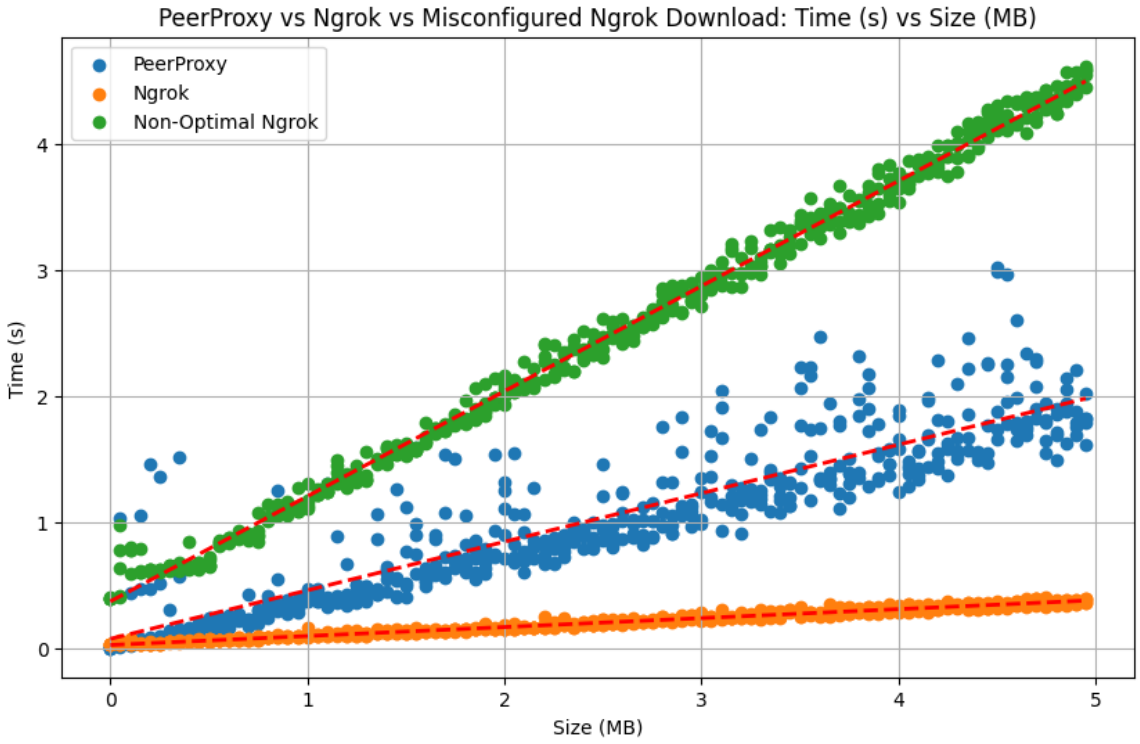


Figure 5.1: Download Throughput Comparison

Table 5.1: Download Throughput Summary

	Download (MB/s)	Base Latency (ms)
PeerProxy	**2.60**	**79.3**
Ngrok	14.08	30.63
Non-Optimal Ngrok	1.2	376

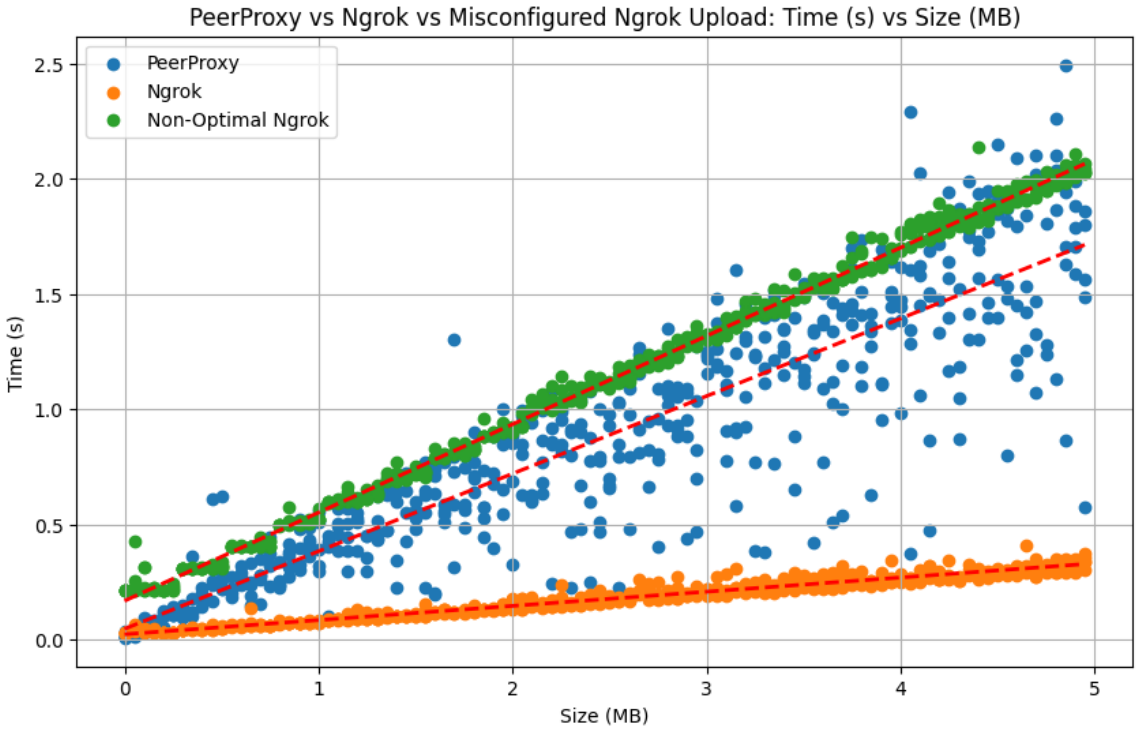


Figure 5.2: Upload Throughput Comparison

Table 5.2: Upload Throughput Summary

	Upload (MB/s)	Base Latency (ms)
PeerProxy	**2.97**	**46.1**
Ngrok	16.23	22.7
Non-Optimal Ngrok	2.6	168

Ngrok, configured correctly, outperforms PeerProxy significantly in both upload (table 5.2) and download (table 5.1). For example, with the download test, Ngrok has a throughput of 14.08 MB/s compared to 2.6 MB/s and even has a lower base latency of 30.63 compared to 79.3. This is an unintuitive result since PeerProxy's direct connection should be faster than a connection made with an intermediate server. An explanation for this is explored more in section 5.1.3.

However, PeerProxy outperforms Ngrok when it is configured non-optimally. As described in section 4.1.1, this is the configuration where the Ngrok proxy server is in Europe. Traffic would travel from the US to Europe back to the US. As shown in table 5.1, PeerProxy had a higher throughput of 2.6 MB/s compared to 1.2 MB/s and a significantly better base latency of 79.3 ms compared to 376 ms. The base latency makes sense since there is a much higher round-trip time, which also decreases throughput.

This could be beneficial to a service provider who only has infrastructure in a certain region. As described in section 4.1.1, relying on a centralized proxy infrastructure in a single region can lead to significant performance issues for users far from that region, increasing round-trip times and reducing throughput. This makes PeerProxy useful for service providers with regionally limited infrastructure, lowering round trip times and increasing performance in some cases.

5.1.2 Latency

The latency tests were designed to measure the time taken to process and complete requests. Each test was run in blocks of 100 to 500 requests, with each block requesting data of varying sizes, from 0 to 1 MB. Like the throughput tests, the latency tests involved downloading data with the Fetch API. Unlike the throughput tests, the requests were executed concurrently, with multiple requests done at the same time to simulate a more realistic load.

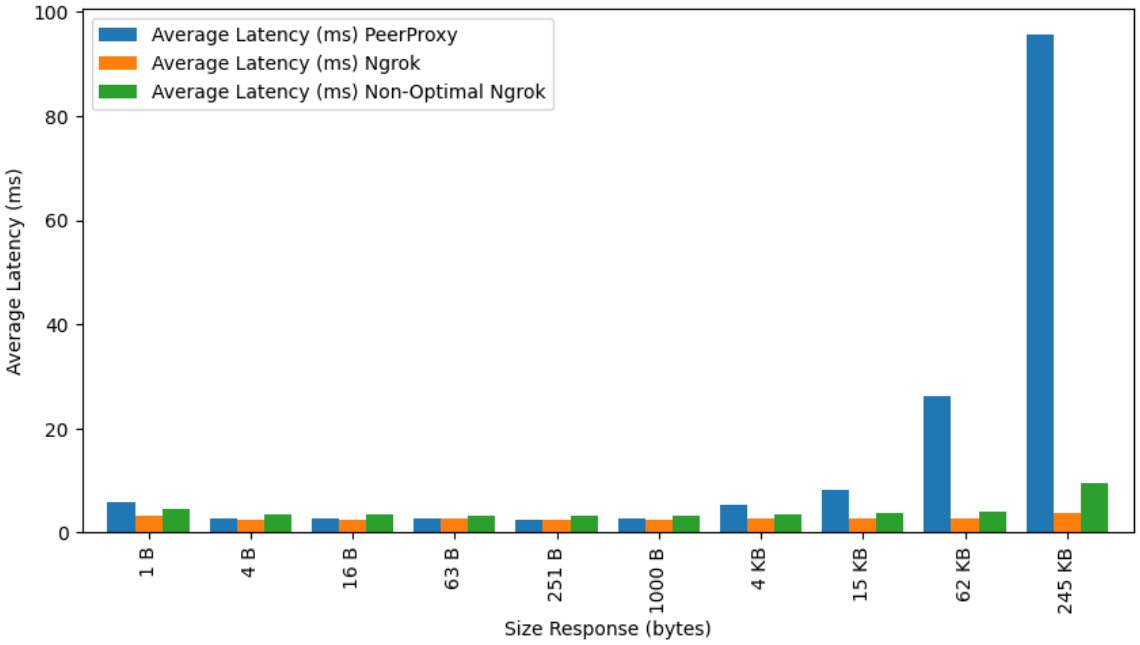


Figure 5.3: Average Latency Comparison

Table 5.3: Latency Summary

Size Response	**PeerProxy (ms)**	Ngrok (ms)	Non-Optimal Ngrok (ms)
1 B	**5.83**	3.37	4.63
4 B	**2.74**	2.47	3.46
16 B	**2.78**	2.49	3.46
63 B	**2.75**	2.74	3.22
251 B	**2.60**	2.44	3.20
1 KB	**2.76**	2.44	3.33
4 KB	**5.31**	2.80	3.61
15 KB	**8.26**	2.70	3.73
62 KB	**26.32**	2.66	4.09
245 KB	**95.69**	3.70	9.40

As shown in figure 5.3 and table 5.3, PeerProxy has similar performance to Ngrok for small payloads up to around 1 KB. However, PeerProxy exhibits significantly higher latencies for larger payloads. As discussed in section 5.1.1, PeerProxy has higher throughput than non-optimal Ngrok, but still has much higher latency for larger requests.

Table 5.4: Latency Test Throughput

Size Response	**PeerProxy (MB/s)**	Ngrok (MB/s)	Non-Optimal Ngrok (MB/s)
1 B	**0.0002**	0.0003	0.0002
4 B	**0.0015**	0.0016	0.0012
16 B	**0.0058**	0.0064	0.0046
63 B	**0.0229**	0.0230	0.0196
251 B	**0.0965**	0.1029	0.0784
1 KB	**0.3623**	0.4098	0.3003
4 KB	**0.7497**	1.4218	1.1028
15 KB	**1.9188**	5.8700	4.2491
62 KB	**2.3973**	23.7203	15.4269
245 KB	**2.6250**	67.8889	26.7222

Table 5.4 shows the average throughput from the latency test for each sized response. This is done by dividing the average time for response by the size of the data. For response sizes of 62 KB and 245 KB, Ngrok's throughput significantly exceeds the 14.08 MB/s measured in Section 5.1.1, reaching 23.7 MB/s (1.7x higher) for 62 KB and 67.89 MB/s (4.8x higher) for 245 KB.

This discrepancy can be explained by examining how TCP and SCTP handle congestion control under sustained loads. TCP has an adaptive congestion control mechanism. That means that when more data is sent over the connection, it increases its congestion window, which allows more data to be in flight. This increases bandwidth utilization and increases throughput. Meanwhile, the SCTP implementation in browsers has a fixed and small congestion window size, limiting the amount of bandwidth utilization and throughput. This is discussed further in section 5.1.3.

So for the smaller sizes, PeerProxy and Ngrok perform similarly since TCP does not expand its congestion window and they both have a similar congestion window. However, as the request sizes increase, TCP increases its congestion window, allowing for far higher throughput, while PeerProxy stays at around the same throughput.

5.1.3 Throughput Performance Bottlenecks

The throughput and latency tests revealed that PeerProxy has a throughput bottleneck. To investigate the underlying cause, tests were conducted similar to the one run in previous work [41]. As discussed in section 2.5.1, the authors found that the data channel implementation built on SCTP was not optimized for high-throughput applications because of a small and fixed SCTP window. Since the paper was published in 2015, the SCTP window in chromium has been increased from 128 KB to 2 MB [102].

The following test was conducted to see if the browser SCTP implementation is still unoptimized for high throughput applications. The test involves sending large amounts of data from the server to a browser in a low-latency environment on the same computer. Then, artificial latency is added to increase RTT using the tool clumsy [56]. The browser client then measures throughput and round-trip time. Round-trip

time is the time it takes for a packet to get from the browser to the server and back to the browser. There were two versions of the test, one with HTTP over TCP and one with WebRTC data channels that use SCTP. In the TCP version, a streaming endpoint was used to send data as fast as the browser could receive it. The WebRTC version was similar, with the server sending the browser data as fast as the browser could accept it, but with a WebRTC data channel.

5.1.3.1 WebRTC Data Channel Throughput vs RTT

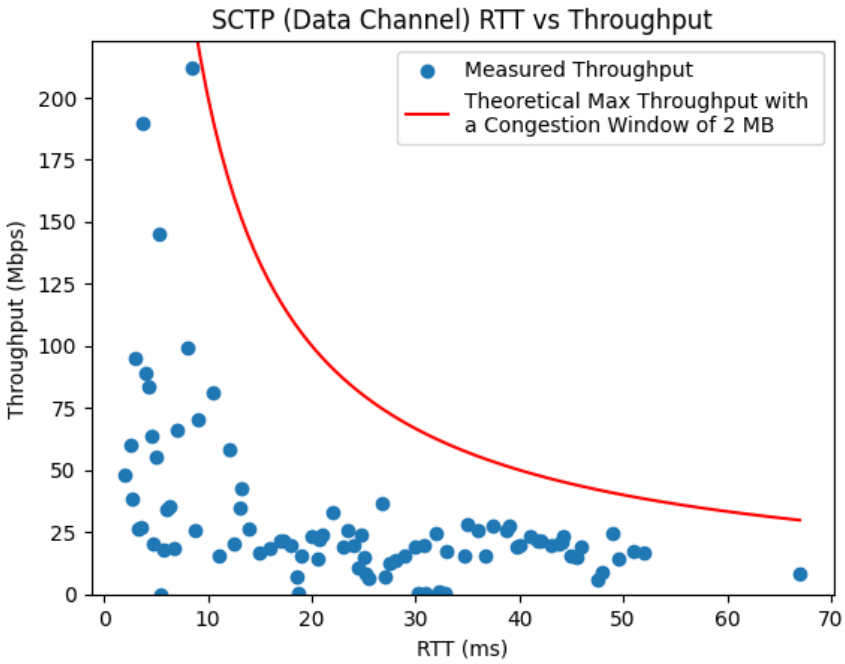


Figure 5.4: Measured Data Channel Throughput at Different RTT with Theoretical Max Throughput

In Figure 5.4, the theoretical max throughput for an SCTP window of 2 MB was plotted with the measured data. This was calculated with the bandwidth-delay-product (BDP) or how large the SCTP window is. The formula for theoretical max throughput is shown below.

$$\text{Max Throughput} = \frac{\text{Window size (BDP)}}{RTT}$$

Figure 5.4 shows the limitations of the SCTP implementation in browsers. The red curve represents an upper bound on the throughput if the sender was able to fully utilize the available window at a given round-trip time. The measured throughput follows a similar shape to the theoretical curve, which suggests that the SCTP implementation uses a fixed send window. That means that the data channel cannot transmit more than what the window allows at a given moment. Even in an ideal scenario, the throughput cannot exceed the theoretical maximum.

5.1.3.2 TCP Throughput vs RTT

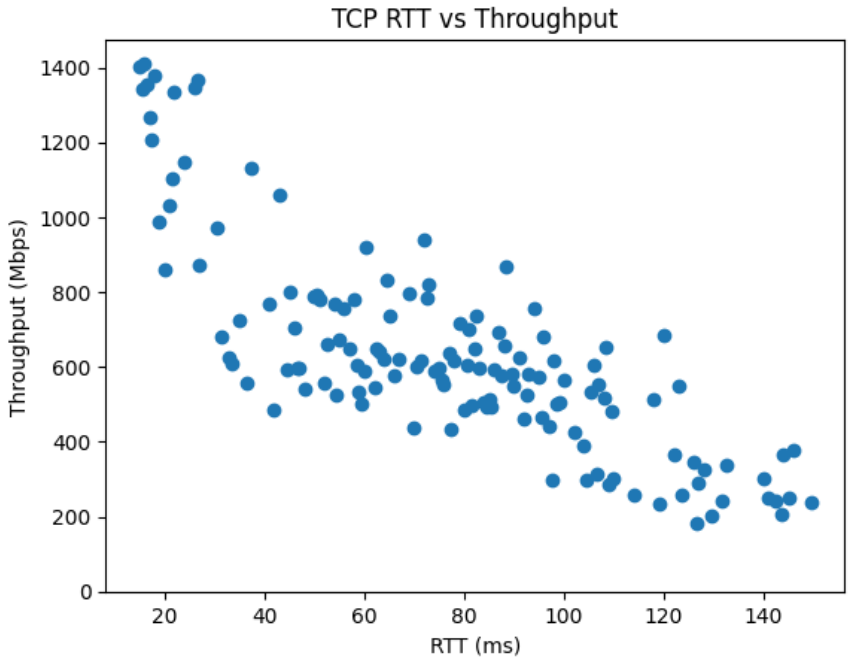


Figure 5.5: TCP Throughput at Different Round Trip Times

Figure 5.5 shows the throughput vs RTT test for TCP. TCP has far higher throughput, up to 1400 MB/s, at 20 ms latency. As RTT is increased, the throughput drops but still sustains much higher throughput than SCTP.

The results of the latency-based tests clearly show that the WebRTC data channel (using SCTP) has much lower throughput than TCP. As illustrated in figure 5.4, throughput for SCTP drops significantly when RTT is increased. This is due to the small fixed window size and latency-optimized congestion control. In contrast, TCP can sustain higher throughput across varying latency conditions while SCTP struggles. TCP has a larger and adaptive window size to better utilize the available bandwidth, allowing it to adjust to varying network situations.

TCP uses a slow start algorithm for congestion control [12]. TCP begins with a small window to minimize latency when data is sparse. As more data is transmitted, TCP rapidly expands its congestion window, allowing a larger volume of data to be in-flight. This dynamic adjustment allows TCP to handle both low and high throughput scenarios while optimizing latency. The latency tests lasted longer than the throughput tests which allowed the congestion window to grow larger and lead to an increase of throughput over time.

In contrast, SCTP uses a fixed window. While this fixed window is effective for low-latency scenarios with minimal data transfer, it becomes a bottleneck in high-throughput applications. The inability to dynamically increase the window means that SCTP cannot fully utilize available network bandwidth, limiting its performance compared to TCP. Implementing dynamic window sizing in SCTP would bridge the performance gap between TCP and SCTP. This would allow SCTP to better use available bandwidth while maintaining low latency.

There is also a proposed experimental alternative to SCTP as a data transport, the QUIC API for peer-to-peer connections [105], which is discussed in section 6.1.1. This protocol is not implemented in modern browsers, but research conducted on an experimental version found it outperforms SCTP in all cases and has similar performance to TCP in some scenarios [99].

This poses a problem for PeerProxy, which relies on SCTP for high-throughput data transfer in current browsers. PeerProxy has comparable performance with respect to third-party proxies for lower throughput applications, but future advancements in the SCTP implementation and new transports would also make it viable for high-throughput applications. PeerProxy should be able to outperform third-party proxies since it forms direct connections without any intermediate servers, but modern browsers' implementation of WebRTC limits it.

5.1.4 Overhead of PeerProxy

The previous section, section 5.1.3, discussed the performance of just a WebRTC data channel. However, PeerProxy is built on WebRTC data channels and introduces additional overhead, such as packet creation and parsing. To evaluate this impact, the throughput vs RTT was tested for PeerProxy and compared to raw WebRTC data channel data.

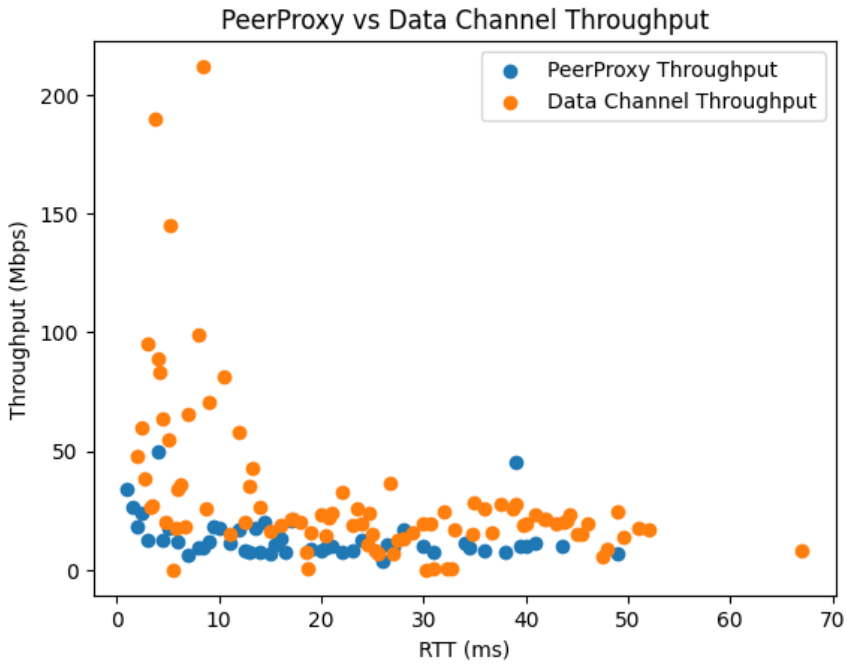


Figure 5.6: PeerProxy introduces minimal overhead to WebRTC data channels

As shown in figure 5.6, PeerProxy has a similar throughput to a data channel above 20ms of RTT but struggles to utilize the higher throughputs of lower RTTs and only reaches 40-50 mb/s. In real-world scenarios where RTT is typically above 20ms, PeerProxy achieves throughput comparable to a direct data channel. This suggests that the additional processing introduced by PeerProxy, such as packet handling and parsing, introduces minimal overhead on top of the WebRTC data channel in most situations. However, if the data channel implementation is improved, as discussed in section 5.1.3, the PeerProxy client could become the bottleneck.

5.2 Resource Utilization Comparison

The following tests compare the CPU and memory usage of the browser and local proxy server running the "latency" benchmark, described in section 5.1.2, through PeerProxy and Ngrok. The "latency" benchmark was selected since it functions as a stress test, issuing network requests as quickly as possible in parallel, maximizing processing usage.

5.2.1 Browser Process Monitoring

The first resource utilization test assessed the browser's overall CPU and memory usage while running the same "latency" benchmark through Ngrok and PeerProxy. To conduct the test, the automation tool Puppeteer was used to open the browser and start the latency test. During the test, the "pidusage" [13] package was used to record CPU and memory usage at 50ms intervals, providing an overview of the browser's resource consumption.

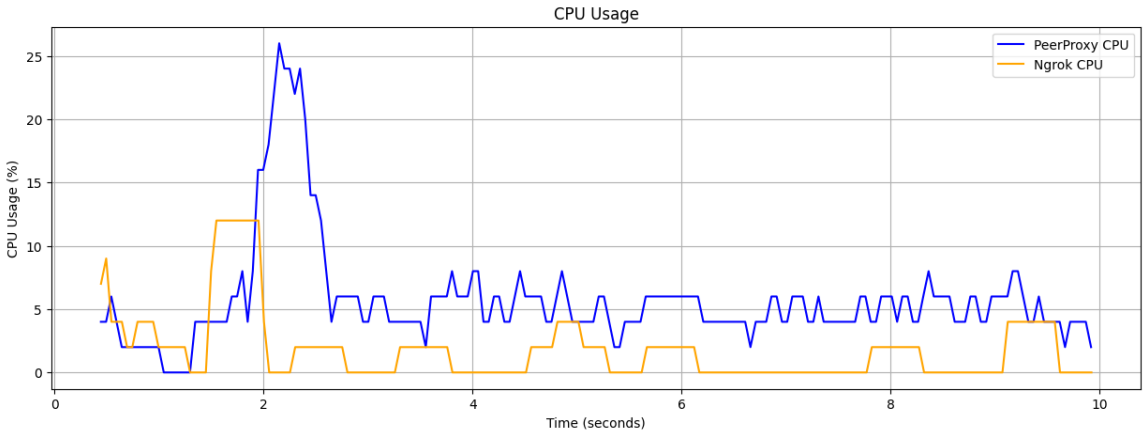


Figure 5.7: CPU Usage in a 10 Second Test

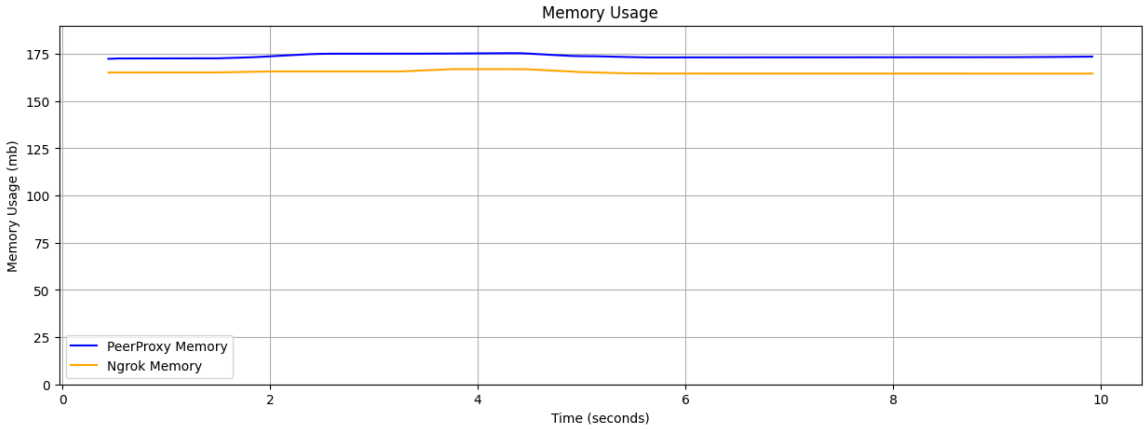


Figure 5.8: Memory Usage in a 10 Second Test

Table 5.5: Comparison of Average CPU and Memory Usage

	Average CPU Usage (%)	Average Memory Usage (mb)
PeerProxy	**5.63**	**173.67**
Ngrok	1.96	165.20

In the summary table 5.5, PeerProxy uses more CPU and averages 5.63% CPU usage, while Ngrok averages 1.96%. As shown in figure 5.8, PeerProxy and Ngrok have similar memory usage, with PeerProxy using an average of 173.7 mb and Ngrok using 165.2 mb.

Browsers were designed from the ground up to efficiently make HTTP requests over traditional transport protocols like TCP [34]. Modern browsers have highly optimized network stacks that minimize latency and CPU overhead when making requests through TCP. Since modern browsers do not support using WebRTC data channels as an HTTP transport, PeerProxy operates outside of this stack. Because of that, PeerProxy has to handle all packet handling manually, including serialization, transmission, and deserialization. This additional processing introduces computational overhead that contributes to PeerProxy's higher CPU and memory usage.

Despite these additional processing steps, PeerProxy still remains a viable way to serve websites. Even with a high workload above what most websites would require, PeerProxy averages 5.63% CPU usage, which is not a significant burden on modern computing hardware. Neither Ngrok and PeerProxy put a substantial load on system resources.

5.2.2 Chrome Performance Tools

For a finer-grained evaluation, this test uses the Chrome DevTools [27] to analyze which portion of each application used resources. These results provided a detailed timeline of CPU and memory usage, giving some insight into potential bottlenecks in PeerProxy as well as overall resource consumption. This test was also run with Puppeteer, using the "tracing" feature [100] to output a Chrome DevTools trace file. Similar to the previous test, this was running the "latency" benchmark.

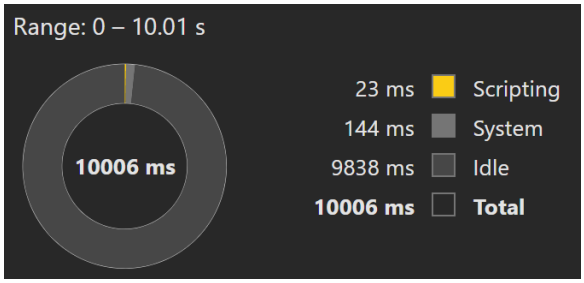
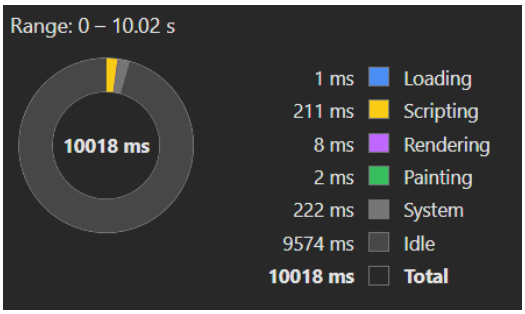
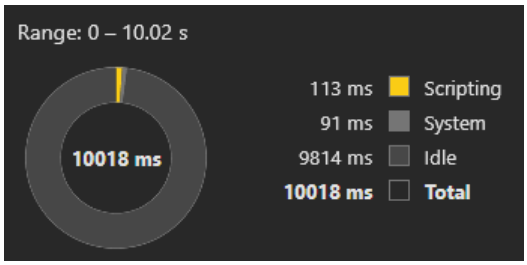


Figure 5.9: Ngrok Scripting Time



(a) PeerProxy Main Thread Scripting Time



(b) PeerProxy Service Worker Scripting Time

Figure 5.10: PeerProxy Scripting Time

Table 5.6: Comparison of Total Scripting Time

	Total Scripting Time
PeerProxy	**324**
Ngrok	23

As shown in table 5.6, PeerProxy has substantially higher scripting time (324 ms) compared to Ngrok (23 ms). This makes sense since PeerProxy relies heavily on client-side scripting to handle all proxying logic and packet processing. In contrast, Ngrok utilizes the browser's native networking stack, offloading much of the processing to the underlying system.

Figure 5.10 shows the amount of scripting time in the main thread and service worker. The main thread is where a browser responds to user events in Javascript, renders the DOM, and lays out elements. By default, the browser uses a single thread per page [68]. Since PeerProxy and the proxied webpage both use the main thread, it's important for PeerProxy to minimize using the main thread so that the loaded website has more resources to respond to user events and render the page.

The service worker runs on a different thread, so execution there does not block the main thread. Figure 5.10b shows that PeerProxy has a decent amount of scripting time in the service worker (113 ms) of the total 324 ms. The scripting time on the main thread (211 ms) does reduce the amount of responsiveness of a proxied website but by a very small margin.

Although PeerProxy's scripting time is significantly higher than Ngrok's, both values are relatively small compared to the total execution time of 10,000 ms. Under a workload higher than what most websites would require, PeerProxy's scripting accounts for 2.11% of the total time, while Ngrok's scripting is just 0.23%. These both should not noticeably impact the performance of a proxied website.

5.3 Local Proxy Evaluation

Each proxy tested has its own local proxy server making requests to the original server. Ngrok's local proxy server connects to intermediate proxy servers, while PeerProxy's local proxy server acts as an adapter for HTTP over WebRTC (described more in detail in section 3.3). This test evaluated each local proxy server's CPU and memory usage under load. The latency test makes many concurrent requests to the local proxy server, so this was used to load the local proxy servers.

To measure the resource usage, a Node.js script spawned the local proxy server as a subprocess. In the script, the library "pidusage"[23] was used to record the CPU and memory usage every 50ms. Since the Ngrok local proxy server spawns multiple processes, the library "pidtree" [88] was used to keep track of processes. The CPU usage results are a sum of the utilization of multiple cores, so they can be over 100%. These tests were run with the entire setup on one computer to increase throughput, maximizing utilization of the local proxy server.

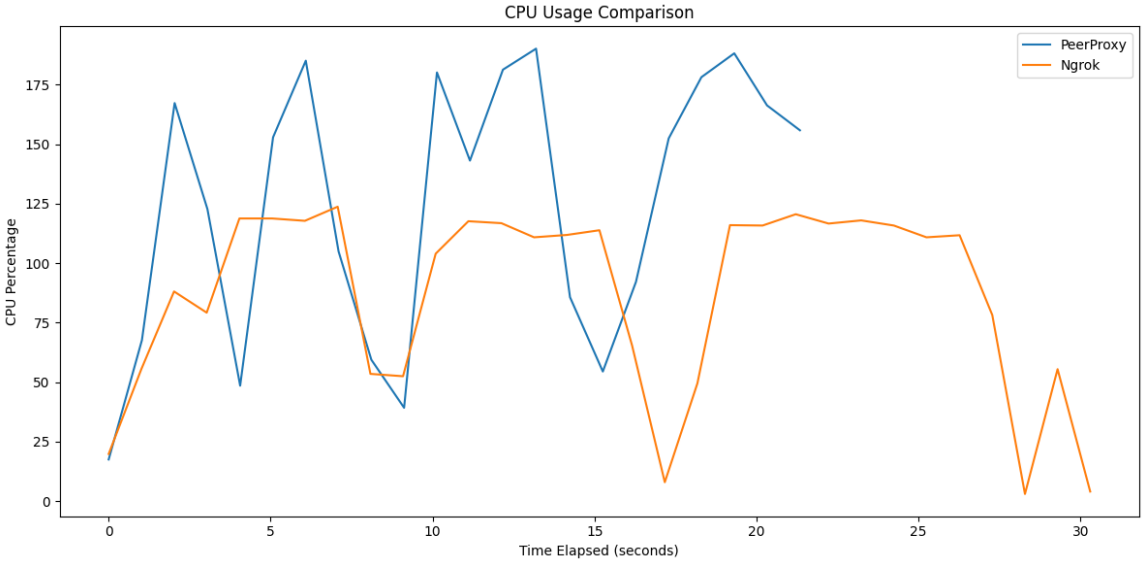


Figure 5.11: Local Proxy Server CPU Comparison

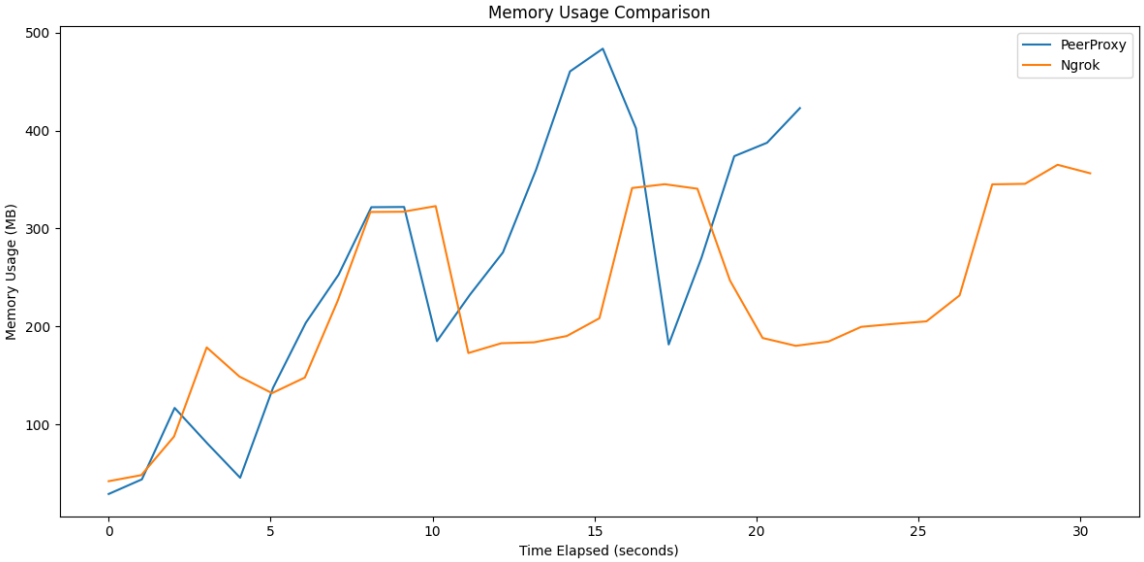


Figure 5.12: Local Proxy Server Memory Comparison

Table 5.7: Comparison of Average CPU and Memory Usage

	Time Elapsed (s)	CPU Usage (%)	Memory Usage (MB)
PeerProxy	**21.34**	**124.26**	**253.98**
Ngrok	30.3	86.83	225.25

As shown in table 5.7, due to different throughputs, the PeerProxy and Ngrok tests processed the same amount of requests but did not run for the same duration. PeerProxy completed the test in a shorter time (21.34 s) but with higher resource usage (124.26% CPU and 253.98 MB memory), while Ngrok ran for a longer time (30.3 s) but with lower resource usage (86.3% CPU and 225.25 MB memory). This makes direct comparisons of average CPU and memory somewhat misleading, as they don't account for the total amount of computational work performed.

To normalize the result over time, the area under the curve for CPU and memory can be calculated from figures 5.11 and 5.12. This gives us a measure of total resource consumption over time, which we can measure as CPU-seconds and Memory-seconds. This was calculated using the Numpy trapezoidal rule function [80].

Table 5.8: Comparison of Total CPU and Memory Work

	Memory Work (MB-seconds)	**CPU Work (CPU-seconds)**
PeerProxy	**5454.37**	**2689.75**
Ngrok	6848.93	2708.48

As shown in table 5.8, PeerProxy and Ngrok have a similar total CPU work, indicating that they both performed a comparable amount of processing over time. However, PeerProxy has a lower total memory work compared to Ngrok, suggesting that it completed the task while consuming less overall memory. However, in a real-world scenario, the difference in total resource consumption may be small enough that it doesn't have a significant impact on overall system performance.

5.4 Connection Time

As outlined in section 2.1.4, establishing a WebRTC connection takes more time than a typical TCP connection. The peers have to connect to an intermediate signaling server, and ICE candidates have to be gathered and exchanged. This adds initial overhead when loading a website through PeerProxy.

To quantify this overhead, the PeerProxy source code was modified to time the connection time and output it to the browser console. A Puppeteer script was used to automate the testing process. For each test, Puppeteer opened a new tab, went to the proxy URL of the Host Server, and recorded the connection time from the console. This process was repeated 10 times to get an average connection establishment time.

This test was conducted with the two EC2 servers located in the same region. However, the servers are located in different subnets and form a connection with NAT traversal using server reflexive candidates (more details are discussed in section 2.1.3.2. This is the most common type of WebRTC connection and makes up around 82% of WebRTC connections [49].

Table 5.9: Measured Connection Times (ms)

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
648.4	479.0	440.1	452.4	438.0	435.7	512.9	441.5	470.9	509.4

Table 5.9 shows connection times over 10 trial runs. They had an average of **482.83ms**. However, both devices were in optimal network conditions in a data center in the same region. Real-world connection times are likely to be longer due to network variability or geographical distance between peers. A study conducted on real-world data of WebRTC connections on the platform *appear.in* showed a median connection time of 1100 ms [1].

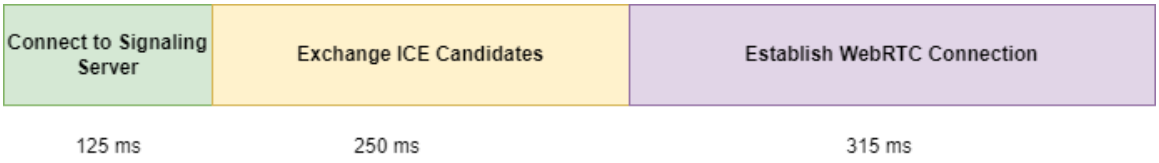


Figure 5.13: WebRTC Connection Timeline

To dive deeper into what expected connection times would be, figure 5.13 shows a breakdown of the timing of each event during a connection of one of the trial runs in table 5.9. The categories "Connect to Signaling Server" and "Exchange ICE Candidates" are likely to remain constant since those involve a straightforward WebSocket request. However, the "Establish Connection" step after exchanging candidates can be more variable due to ICE candidate selection and NAT traversal mechanisms.

This connection time is much longer than establishing a TCP connection because the client has to connect to the signaling server, find and exchange ICE candidates, and find an optimal connection path. However, as discussed in section 3.7, PeerProxy has some optimizations where the WebRTC connection is kept open through page navigations. This mitigates some of the impact when a user is navigating between web pages but still has an initial long connection time.

5.5 Device Compatibility

5.5.1 Browser Client

There are many different web browsers used worldwide, and they all support various technologies and standards. The PeerProxy browser client uses several advanced browser APIs, such as WebRTC and Service Workers, so it isn't supported by all web browsers.

The tool browserslist was used to assess the global percentage of browsers capable of running PeerProxy. Browserslist combines browser market share data with supported API data to determine what percentage of users globally can use a certain site [20]. The static code analysis tool eslint-plugin-compat [22] was run on the browser client codebase to check that the APIs used were compatible with browsers specified by browserslist.

As of August 2024, PeerProxy is compatible with **82.6%** of browsers globally, including modern desktop and mobile versions of Chrome, Firefox, Edge, and Safari.

5.5.2 Local Server Proxy

The local server proxy can run anywhere that Go compiles to. Go has extensive cross-compilation support and can easily compile to popular platforms such as Linux, Windows, and Mac with support for major CPU architectures [8].

5.6 Unsupported Browser APIs

Since PeerProxy loads a website in a non-standard way, a proxied website cannot use a few browser APIs. The PeerProxy browser client cannot support or polyfill service workers or same-domain iframes, but support can be added in the future for cookies and websockets discussed in section 6.1.2.

Service Workers: Service workers are a browser API that intercepts and modifies requests from a webpage. There can only be one service worker registered per page, and PeerProxy already has one.

Cookies: Cookies are small pieces of information sent by a server to a browser to be stored [10]. These are commonly used for managing user sessions and website personalization. Due to security concerns, browsers do not let Javascript scripts add cookies to HTTP responses [67]. PeerProxy manually constructs Response objects to send back to the webpage, so it isn't able to add cookies to the response to be stored in the browser. This is a large compatibility issue since most web authentication techniques use cookies. However, support can be added and is discussed in section 6.1.2.2.

Same Domain Websockets: Service Workers only intercept normal HTTP network requests and do not intercept WebSocket requests. If a proxied webpage tries to establish a same-domain connection, it will fail. Support can be added for this and is discussed in section 6.1.2.1.

5.7 Summary of Findings

This evaluation of PeerProxy demonstrates that it is a viable alternative to traditional HTTP proxies in many scenarios but has certain tradeoffs.

PeerProxy offers reasonable throughput and latency for most applications. However, its performance is constrained by the WebRTC data channel, which has a fixed SCTP window size and unoptimized congestion control, limiting throughput. Tests in this section show that PeerProxy achieves a download throughput speed of 2.6 MB/s, significantly lower than Ngrok, which has a download throughput of 14.08 MB/s. However, PeerProxy has better throughput performance compared to a setup where Ngrok is configured non-optimally, which could be a benefit to service providers who don't have global proxying infrastructure. Additionally, PeerProxy exhibits higher latency for larger payloads, particularly beyond 1 KB, where Ngrok outperforms it. This is due to WebRTC's congestion control mechanisms prioritizing low latency over high throughput, making it less efficient for large data transfers. There is a proposed standard for adding a new data transport to WebRTC that outperforms the current data channel and has similar performance to TCP in some scenarios, further discussed in section 6.1.1.

Establishing a WebRTC connection introduces additional latency over normal TCP connections. These connection times range from 482.8 ms in optimal conditions to 1100 ms in real-world scenarios. PeerProxy has some workarounds to keep the WebRTC connection open over page navigations but still has the initial overhead. This is a notable tradeoff compared to traditional TCP-based proxies and can affect usability.

The PeerProxy browser client also maintains broad compatibility with modern web browsers, supporting approximately 82.6% of global users. PeerProxy is compatible with the latest versions of Chrome, Firefox, Edge, and Safari. The PeerProxy browser client uses more CPU and memory and more scripting time on the main thread running the latency benchmark than Ngrok, but it should not noticeably impact the performance of proxied websites.

The PeerProxy local proxy server is also compatible with all common operating systems and architectures. PeerProxy's local proxy server also has similar or better performance to the Ngrok local proxy server. The PeerProxy browser client can proxy websites that do not rely on service workers or same-domain iframes. Support can be added for websockets and cookies in future versions of PeerProxy, and is discussed in 6.1.2.

PeerProxy presents a compelling alternative to centralized proxy solutions like Ngrok with direct peer-to-peer connections, enhancing privacy for users and lowering costs for service providers. However, it has some tradeoffs, such as reduced performance and increased connection time. Future work should explore adding support for websockets and cookies and evaluating alternative transport protocols.

Chapter 6

CONCLUSION

6.1 Future Work

6.1.1 Future Data Transport for WebRTC

As discovered in section 5.1.3, PeerProxy is bottlenecked by the browser data channel implementation not being able to handle high throughput. There's a proposed API to use the new WebTransport protocol over a peer connection. WebTransport is a newer protocol that facilitates low-latency, bidirectional communication between servers and clients. It's similar to WebSockets but offers better performance, reliability, and low-level control. As of October 2024, the feature is supported in all modern browsers except Safari [65].

However, there's a W3C proposal for using WebTransport as an alternative data transport for WebRTC with the RTCQuicTransport API [105]. This was briefly implemented in two versions of Chrome in 2019 as an origin trial to gather feedback [18]. Since this was barely implemented, there have been limited performance evaluations, but it looks promising so far. In a memo published by the live streaming company System 73, the authors benchmarked the throughput of SCTP, RTCQuicTransport, and TCP with different communication latencies. The authors found that RTCQuicTransport outperformed SCTP in all cases and had similar results to TCP in some scenarios [99].

However, there is no guarantee that this will ever be implemented as a web standard. But if it ever is, it could significantly improve the performance of PeerProxy and other high-throughput WebRTC applications. This could make PeerProxy and similar applications rival or surpass the performance of third-party proxies.

6.1.2 Add Support For More Web Features

As described in section 5.6, some browser APIs are unsupported in the base version of PeerProxy but can be supported in future versions.

6.1.2.1 WebSockets

The Service Worker only supports intercepting HTTP requests and not WebSocket requests. In the current implementation, if the proxied site makes a WebSocket request to an address on the same domain, it will fail since there's nothing to handle the request.

Since the browser client has some control over the sandbox, one solution is to override the default WebSocket object with a custom one. This would implement the same methods so that proxied websites can work without modification. Another custom protocol similar to HTTP over WebRTC would have to be developed for WebSocket to manage WebSocket connections. This feature would be particularly valuable for IoT solutions, where efficiently receiving real-time data is essential.

6.1.2.2 Cookies

Cookies aren't supported since browsers don't let JavaScript manage cookies for security reasons. However, adding support for cookies in PeerProxy would be valuable. Exposing a web service directly to the Internet could come with security risks, so some authentication must be done. Most browser authentication techniques use cookies, so this would make existing sites more compatible and make adding authentication to new sites easier. A good and secure solution will require more investigation.

6.2 Contributions

This thesis makes several contributions, including a functional prototype of PeerProxy, a thorough evaluation of its performance, and the current state of building high-throughput applications on WebRTC.

6.2.1 A Functioning Prototype of PeerProxy

This thesis presents a prototype of PeerProxy, demonstrating the feasibility of serving web applications with WebRTC. The prototype successfully integrates all components, such as a browser client, local proxy server, and signaling server. These components are fully functional and can proxy a web server to anywhere for anyone. PeerProxy also offers a user experience comparable to that of commercial proxies. To proxy a web server, a user runs the PeerProxy local proxy server binary and gets a link that anyone can use to view web pages without any networking configuration. This provides a more private and infrastructure-efficient alternative to using a commercial proxy such as Ngrok.

The browser client demonstrates that adding WebRTC as a transport mechanism for HTTP in an unmodified modern browser is possible. The client uses a combination of existing web technologies, such as service workers and iframes, to seamlessly intercept, serialize, and deliver HTTP requests without modification to the browser or the proxied website. It does this while maintaining support with 82.6% of browsers globally.

PeerProxy also presents a custom packet protocol for proxying HTTP traffic using WebRTC data channels. This protocol can load and stream data of arbitrary size with minimum overhead.

6.2.2 Performance Evaluation

This work also thoroughly evaluates PeerProxy's performance, examining latency, throughput, connection times, and resource utilization of the browser client and local proxy server. These tests compared the performance of PeerProxy and Ngrok and revealed how PeerProxy performs in the real world and what applications it's suitable for. PeerProxy is limited by current browsers' implementation of WebRTC data channels. However, if there's further optimization down the line or a new protocol, having this framework to evaluate PeerProxy will help guide and evaluate future optimizations.

6.2.3 Investigating High Throughput Applications Using WebRTC

The thesis also tests using WebRTC data channels for high-throughput applications by examining their performance under different network conditions. The evaluation shows that WebRTC's SCTP implementation is not optimized for consistently high throughput, especially when there's a higher round-trip time. This evaluation will help benchmark and compare future implementations of WebRTC data channels in the browser.

6.3 PeerProxy Applications

PeerProxy has several promising applications, particularly in scenarios with privacy concerns or restrictive network situations.

6.3.1 Self-Hosted Applications

As outlined throughout this work, a natural application for PeerProxy is with self-hosted applications. Traditional methods of making self-hosted services available externally often require port forwarding, which can be technically challenging or not possible in certain network situations. PeerProxy simplifies this by eliminating the need for manual network configuration, enabling direct connections between the user's browser and a self-hosted service.

The alternative to port forwarding is using proxy-based services with a publicly accessible server that forwards traffic to the self-hosted service. However, these proxies introduce additional concerns, such as increased latency, potential data privacy risks due to third-party access, and ongoing operational costs. PeerProxy addresses these

challenges by using end-to-end encrypted connections, enhancing security and using direct peer-to-peer connections around 82% of the time [49] for reduced operational overhead. This allows users to maintain full control over their data and infrastructure while easily and securely accessing their self-hosted services remotely.

6.3.2 Internet of Things (IoT)

The Internet of Things or IoT describes a network of devices that communicate and exchange data over the internet [52]. From a consumer standpoint, these are devices like smart thermostats, security cameras, and lighting systems. These devices often need remote management or data access, and this is typically done through centralized proxies, which introduces infrastructure overhead and privacy concerns. IoT companies using a framework like PeerProxy can significantly reduce infrastructure costs since most connections can be made directly and do not require relay servers [49]. Direct connections also lower the need for globally distributed proxy servers that minimize routing distance, as described in section 5.1.1, further driving down infrastructure costs. IoT data can also be particularly sensitive since it often includes highly personal information that can be a prime target for unauthorized access or exploitation. A framework like PeerProxy ensures end-to-end encryption, which enhances user privacy, as data never passes through unencrypted third-party servers, mitigating the risk of unauthorized access or data breaches.

6.3.3 Developer Application Testing

Developers frequently need a simple way to expose local development servers to clients, collaborators, or external testing environments. However, developers are often behind enterprise networks, which often have more restrictive network settings [72].

PeerProxy simplifies this process by eliminating the need for cumbersome firewall adjustments or port forwarding setups. It allows rapid sharing of local development environments through secure, encrypted tunnels. This capability accelerates collaboration and iterative testing, providing developers with immediate, secure feedback from external users without compromising corporate network security or requiring administrative network modifications.

6.4 Final Thoughts

PeerProxy explores whether it's possible to serve websites directly from behind local networks. With current web technologies, it is possible, and PeerProxy is viable for some applications. However, because of current limitations in web technology, PeerProxy trades off decreased performance for increased privacy for users and reduced infrastructure costs for service providers.

With potential future advancements like RTCQuicTransport or SCTP optimizations, PeerProxy could outperform commercial proxies since it forms direct connections with fewer intermediaries to add round-trip time. In the meantime, the tests developed in this thesis can be used to evaluate similar systems as these technologies evolve.

Overall, PeerProxy demonstrates that it's possible to decentralize web traffic by removing the reliance on centralized infrastructure. This gives users back control of their data by not sending it through any third parties and encrypting data end-to-end. For service providers, technologies like PeerProxy can significantly reduce proxying infrastructure and lower the need for complex multi-region infrastructure. PeerProxy has demonstrated that building systems in this manner can benefit users and service providers.

BIBLIOGRAPHY

[1] D.-I. Aas. *WebRTC Connection Times and the Power of Playing Around with Data*. https://medium.com/the-making-of-whereby/webrtc-connection-times-and-the-power-of-playing-around-with-data-ab11312737e9. Accessed: 2024-09-30. 2016.

[2] H. Alvestrand. *Google release of WebRTC source code*. https://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html. Accessed: 2024-06-10. 2011.

[3] H. T. Alvestrand. *Overview: Real-Time Protocols for Browser-Based Applications*. RFC 8825. Jan. 2021. DOI: 10.17487/RFC8825. URL: https://www.rfc-editor.org/info/rfc8825.

[4] I. Amazon Web Services. *Amazon EC2 - Elastic Compute Cloud*. https://aws.amazon.com/ec2/. Accessed: 2024-08-09. 2024.

[5] I. Amazon Web Services. *Amazon EC2 Instance Types*. https://aws.amazon.com/ec2/instance-types/. Accessed: 2024-08-09. 2024.

[6] I. Amazon Web Services. *Amazon Web Services (AWS)*. https://aws.amazon.com/. Accessed: 2024-08-09. 2024.

[7] Anthony Sidashin. *Self-Hosted is Awesome*. https://pixeljets.com/blog/self-hosted-is-awesome/. Accessed: 2025-02-07. 2024.

[8] K. Asuka. *Go (Golang) GOOS and GOARCH*. https://gist.github.com/asukakenji/f15ba7e588ac42795f421b48b8aede63. Accessed: 2024-08-09. 2024.

[9] D. Barradas, N. Santos, L. Rodrigues, and V. Nunes. "Poking a Hole in the Wall: Efficient Censorship-Resistant Internet Communications by Parasitizing on WebRTC". In: *Proceedings of the 2020 ACM SIGSAC Conference on*

Computer and Communications Security. CCS '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 35–48. ISBN: 9781450370899. DOI: 10.1145/3372297.3417874. URL: https://doi.org/10.1145/3372297.3417874.

[10] A. Barth. *HTTP State Management Mechanism*. RFC 6265. Apr. 2011. DOI: 10.17487/RFC6265. URL: https://www.rfc-editor.org/info/rfc6265.

[11] A. C. Begen, P. Kyzivat, C. Perkins, and M. J. Handley. *SDP: Session Description Protocol*. RFC 8866. Jan. 2021. DOI: 10.17487/RFC8866. URL: https://www.rfc-editor.org/info/rfc8866.

[12] E. Blanton, D. V. Paxson, and M. Allman. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI: 10.17487/RFC5681. URL: https://www.rfc-editor.org/info/rfc5681.

[13] A. Bluchet and S. Primarosa. *pidusage: Cross-platform process CPU and memory usage of a PID*. https://www.npmjs.com/package/pidusage. Version 3.0.2. 2023.

[14] N. Blum, S. Lachapelle, and H. Alvestrand. "WebRTC - Realtime Communication for the Open Web Platform: What was once a way to bring audio and video to the web has expanded into more use cases we could ever imagine." In: *Queue* 19.1 (Mar. 2021), pp. 77–93. ISSN: 1542-7730. DOI: 10.1145/3454122.3457587. URL: https://doi.org/10.1145/3454122.3457587.

[15] C. Bocovich, A. Breault, D. Fifield, X. Wang, et al. "Snowflake, a censorship circumvention system using temporary {WebRTC} proxies". In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 2635–2652.

[16] S. O. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. Mar. 1997. DOI: 10.17487/RFC2119. URL: https://www.rfc-editor.org/info/rfc2119.

[17] S. Cheshire and M. Krochmal. *Multicast DNS*. RFC 6762. Feb. 2013. DOI: 10.17487/RFC6762. URL: https://www.rfc-editor.org/info/rfc6762.

[18] Chrome Developers. *RTCQuicTransport Coming to an Origin Trial Near You (Chrome 73)*. https://developer.chrome.com/blog/rtcquictransport-api. Accessed: 2024-10-06. 2019.

[19] I. Cloudflare. *Cloudflare Tunnel - Securely expose your local server to the internet*. https://www.cloudflare.com/products/tunnel/. Accessed: 2024-08-09. 2024.

[20] B. Community. *Browserslist - Shared browser compatibility configuration*. https://browsersl.ist/. Accessed: 2024-08-09. 2024.

[21] M. Community. *MessagePack - It's like JSON, but fast and small*. https://msgpack.org/index.html. Accessed: 2024-08-09. 2024.

[22] N. Community. *eslint-plugin-compat*. https://www.npmjs.com/package/eslint-plugin-compat. Accessed: 2024-08-09. 2024.

[23] N. Community. *pidusage*. https://www.npmjs.com/package/pidusage. Accessed: 2024-08-09. 2024.

[24] P. Community. *Puppeteer Documentation*. https://pptr.dev/. Accessed: 2024-08-09. 2024.

[25] P. Contributors. *PeerJS - Simple peer-to-peer with WebRTC*. https://peerjs.com/. Accessed: 2024-07-02. 2024.

[26] ws Contributors. *ws: a Node.js WebSocket library*. Version 8.18.0. 2024. URL: https://www.npmjs.com/package/ws.

[27] G. Developers. *Chrome DevTools Performance Reference*. https://developer.chrome.com/docs/devtools/performance/reference. Accessed: 2024-08-09. 2024.

[28] Digital Samba. *The Power of E2EE in WebRTC: Unlocking a World of Secure Communication*. https://www.digitalsamba.com/blog/the-power-of-e2ee-in-webrtc-unlocking-a-world-of-secure-communication. Accessed: 2025-02-07. 2024.

[29] Docker, Inc. *Docker*. https://www.docker.com/. Accessed: 2025-03-02. 2025.

[30] M. W. Docs. *¡iframe¿ - HTML: HyperText Markup Language*. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe. Accessed: 2024-08-09. 2024.

[31] M. W. Docs. *Cache - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/Cache. Accessed: 2024-08-09. 2024.

[32] M. W. Docs. *Document.write() - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/Document/write. Accessed: 2024-08-09. 2024.

[33] M. W. Docs. *HTTP as an application layer protocol, on top of TCP (transport layer) and IP (network layer) and below the presentation layer.* https://mdn.github.io/shared-assets/images/diagrams/http/overview/http-layers.svg. Accessed: 2024-08-09. 2024.

[34] M. W. Docs. *HTTP Overview - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. Accessed: 2024-08-09. 2024.

[35] M. W. Docs. *RTCPeerConnection - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection. Accessed: 2024-06-10. 2024.

[36] M. W. Docs. *RTCPeerConnection.getStats() - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/getStats. Accessed: 2024-08-09. 2024.

[37] M. W. Docs. *Service Worker API - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API. Accessed: 2024-08-09. 2024.

[38] M. W. Docs. *Signaling and video calling*. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling. Accessed: 2024-07-25. 2024.

[39] M. W. Docs. *WebSockets API - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. Accessed: 2024-08-09. 2024.

[40] Z. Durumeric, Z. Ma, D. Springall, et al. "The Security Impact of HTTPS Interception". In: *Network and Distributed System Security Symposium (NDSS) 2017*. Accessed: 2025-02-07. 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/security-impact-https-interception/.

[41] R. Eskola and J. K. Nurminen. "Performance evaluation of WebRTC data channels". In: *2015 IEEE Symposium on Computers and Communication (ISCC)*. 2015, pp. 676–680. DOI: 10.1109/ISCC.2015.7884873.

[42] Express.js. *Express - Node.js web application framework*. https://expressjs.com/. Accessed: 2024-08-09. 2024.

[43] Federal Trade Commission. *FTC Says Ring Employees Illegally Surveilled Customers, Failed to Stop Hackers from Taking Control of Users' Cameras*. https://www.ftc.gov/news-events/news/press-releases/2023/05/ftc-says-ring-employees-illegally-surveilled-customers-failed-stop-hackers-taking-control-users. Accessed: 2024-10-06. 2023.

[44] C. B. A. B. D. Fifield and S. X. Wang. "Snowflake, a censorship circumvention system using temporary WebRTC proxies". In: ().

[45] G. Figueira, D. Barradas, and N. Santos. "Stegozoa: Enhancing WebRTC Covert Channels with Video Steganography for Internet Censorship Circumvention". In: May 2022. DOI: 10.1145/3488932.3517419.

[46] Fly.io. *Fly.io - The Platform for Running Full-Stack Apps and Databases*. https://fly.io/. Accessed: 2024-08-09. 2024.

[47] O. Foundation. *Manipulator-in-the-middle attack*. https://owasp.org/www-community/attacks/Manipulator-in-the-middle_attack. Accessed: 2024-08-09. 2024.

[48] A. O. Freier, P. Karlton, and P. C. Kocher. *The Secure Sockets Layer (SSL) Protocol Version 3.0*. RFC 6101. Aug. 2011. DOI: 10.17487/RFC6101. URL: https://www.rfc-editor.org/info/rfc6101.

[49] P. Hancke. *What kind of TURN server is being used?* https://medium.com/@fippo/what-kind-of-turn-server-is-being-used-d67dbfc2ff5d. Accessed: 2024-10-06. 2020.

[50] HAProxy Technologies. *What is SSL/TLS Termination?* https://www.haproxy.com/glossary/what-is-ssl-tls-termination. Accessed: 2025-02-07. 2024.

[51] M. Holdrege and P. Srisuresh. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 2663. Aug. 1999. DOI: 10.17487/RFC2663. URL: https://www.rfc-editor.org/info/rfc2663.

[52] IBM. *Internet of Things (IoT) - IBM Think*. https://www.ibm.com/think/topics/internet-of-things. Accessed: 2025-03-10. 2025.

[53] *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791.

[54] *IPFS - The InterPlanetary File System*. https://ipfs.tech/. Accessed: 2024-08-08. 2024.

[55] E. Ivov, E. Rescorla, J. Uberti, and P. Saint-Andre. *Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol*. RFC 8838. Jan. 2021. DOI: 10.17487/RFC8838. URL: https://www.rfc-editor.org/info/rfc8838.

[56] Jagt. *clumsy, an utility for simulating broken network for Windows*. Accessed: 2024-10-06. 2024. URL: https://jagt.github.io/clumsy/.

[57] B. Jansen, T. Goodwin, V. Gupta, et al. "Performance Evaluation of WebRTC-based Video Conferencing". In: *SIGMETRICS Perform. Eval. Rev.* 45.3 (Mar. 2018), pp. 56–68. ISSN: 0163-5999. DOI: 10.1145/3199524.3199534. URL: https://doi.org/10.1145/3199524.3199534.

[58] K6. *K6 - Modern Load Testing for Developers*. https://k6.io/. Accessed: 2024-08-09. 2024.

[59] A. Keränen, C. Holmberg, and J. Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. RFC 8445. July 2018. DOI: 10.17487/RFC8445. URL: https://www.rfc-editor.org/info/rfc8445.

[60] T. Leithead. *DOM Parsing and Serialization*. https://www.w3.org/TR/DOM-Parsing/. W3C Working Draft 17 May 2016. 2016.

[61] *libp2p*. https://libp2p.io/. Accessed: 2024-08-08. 2024.

[62] G. LLC. *Google Chrome Web Browser*. https://www.google.com/chrome/. Accessed: 2024-08-09. 2024.

[63] MDN Contributors. *Navigation API*. Accessed: 2024-07-02. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/Navigation_API.

[64] MDN Contributors. *Populating the page: how browsers work*. Accessed: 2024-07-02. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work.

[65] MDN contributors. *WebTransport API*. https://developer.mozilla.org/en-US/docs/Web/API/WebTransport#browser_compatibility. Accessed: 2024-10-06. 2024.

[66] MDN Web Docs. *Fetch API - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. Accessed: 2024-09-30. 2024.

[67] MDN Web Docs. *Forbidden Header Name - MDN Glossary*. https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name. Accessed: 2025-02-07. 2024.

[68] MDN Web Docs. *Main Thread - MDN Glossary*. https://developer.mozilla.org/en-US/docs/Glossary/Main_thread. Accessed: 2024-09-30. 2024.

[69] MDN Web Docs. *RTCDataChannel - Web APIs — MDN*. https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel. Accessed: 2025-02-07. 2024.

[70] A. Melnikov and I. Fette. *The WebSocket Protocol*. RFC 6455. Dec. 2011. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/info/rfc6455.

[71] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. Retrieved 8 October 2022. CRC Press, 1996. Chap. 8: Public-key encryption, pp. 283–319. ISBN: 0-8493-8523-7. URL: https://www.example.com.

[72] Microsoft Learn. *Secure Development Environment - Zero Trust*. https://learn.microsoft.com/en-us/security/zero-trust/develop/secure-dev-environment-zero-trust. Accessed: 2025-03-10. 2025.

[73] MIT Technology Review. *Centralized Web Services Are Wonderful—Until They Go Wrong*. https://www.technologyreview.com/2017/02/24/153563/centralized-web-services-are-wonderful-until-they-go-wrong/. Accessed: 2025-03-02. 2017.

[74] R. Moskowitz, D. Karrenberg, Y. Rekhter, et al. *Address Allocation for Private Internets*. RFC 1918. Feb. 1996. DOI: 10.17487/RFC1918. URL: https://www.rfc-editor.org/info/rfc1918.

[75] M. Moulay and V. Mancuso. "Experimental performance evaluation of WebRTC video services over mobile networks". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2018, pp. 541–546. DOI: 10.1109/INFCOMW.2018.8407020.

[76] MozDevNet. *Using WebRTC Data Channels - Web APIs: MDN*. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Using_data_channels. Accessed: 2024-06-19.

[77] Ngrok. *Ngrok - Secure Tunnels to Localhost*. https://ngrok.com/. Accessed: 2024-08-09. 2024.

[78] ngrok. *Terminate TLS - ngrok Documentation*. https://ngrok.com/docs/traffic-policy/actions/terminate-tls/. Accessed: 2025-03-02. 2024.

[79] H. Nielsen, J. Mogul, L. M. Masinter, et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://www.rfc-editor.org/info/rfc2616.

[80] NumPy Developers. *numpy.trapz — NumPy v1.25 Manual*. https://numpy.org/doc/1.25/reference/generated/numpy.trapz.html. Accessed: 2024-09-30. 2024.

[81] OWASP Foundation. *Cross Site Scripting (XSS)*. Accessed: 2024-10-06. 2024. URL: https://owasp.org/www-community/attacks/xss/.

[82] R. Peon and H. Ruellan. *HPACK: Header Compression for HTTP/2*. RFC 7541. May 2015. DOI: 10.17487/RFC7541. URL: https://www.rfc-editor.org/info/rfc7541.

[83] M. Petit-Huguenin, G. Salgueiro, J. Rosenberg, et al. *Session Traversal Utilities for NAT (STUN)*. RFC 8489. Feb. 2020. DOI: 10.17487/RFC8489. URL: https://www.rfc-editor.org/info/rfc8489.

[84] R. Pike. "Go at Google: Language Design in the Service of Software Engineering". In: (2012). Accessed: 2024-06-22. URL: https://go.dev/talks/2012/splash.article.

[85] Pion WebRTC Contributors. *Pion WebRTC*. Accessed: 2024-06-22. 2024. URL: https://pion.ly/.

[86] PM2.io. *PM2 - Production Process Manager for Node.js*. https://pm2.io/. Accessed: 2024-08-09. 2024.

[87] PortForward.com. *PortForward*. https://portforward.com/. Accessed: 2025-03-02. 2025.

[88] S. Primarosa. *pidtree: Cross platform children list of a PID*. https://www.npmjs.com/package/pidtree. Version 0.6.0. MIT License. 2021.

[89] W. Project. *WebRTC - Real-Time Communication for the Web*. https://webrtc.org/. Accessed: 2024-08-09. 2024.

[90] T. Reddy.K, A. Johnston, P. Matthews, and J. Rosenberg. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 8656. Feb. 2020. DOI: 10.17487/RFC8656. URL: https://www.rfc-editor.org/info/rfc8656.

[91] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012. DOI: 10.17487/RFC6347. URL: https://www.rfc-editor.org/info/rfc6347.

[92] R. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the ACM* 21.2 (Feb. 1978). Archived from the original (PDF) on 2023-01-27, pp. 120–126. DOI: 10.1145/359340.359342. URL: https://www.example.com.

[93] J. Rosenberg, C. Huitema, R. Mahy, and J. Weinberger. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. RFC 3489. Mar. 2003. DOI: 10.17487/RFC3489. URL: https://www.rfc-editor.org/info/rfc3489.

[94] P. C. Sean DuBois. *WebRTC for the Curious*. Accessed: 2024-07-02. 2022.

[95] Socket.IO. *Socket.IO - Real-time application framework*. https://socket.io/. Accessed: 2024-08-09. 2024.

[96] S. G. Stats. *Desktop vs Mobile vs Tablet Market Share Worldwide*. https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet. Accessed: 2024-08-09. 2024.

[97] T. Steeves. "WebRTC NAT Traversal Methods: A Case for Embedded TURN". In: (Mar. 2022). Accessed: 2024-12-07.

[98] R. R. Stewart, M. Tüxen, S. Loreto, and R. Seggelmann. *Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol*. Internet-Draft draft-ietf-tsvwg-sctp-ndata-13. Work in Progress. Internet Engineering Task Force, Sept. 2017. 23 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-tsvwg-sctp-ndata/13/.

[99] System73. *WebRTC QUIC/SCTP comparison*. Proprietary Document. Accessed: 2024-10-06. 2024.

[100] P. D. Team. *Puppeteer Tracing Class API*. https://pptr.dev/api/puppeteer.tracing. Accessed: 2024-09-29. 2024.

[101] TechRepublic. *The Most Important Cloud Advances of the Decade*. https://www.techrepublic.com/article/the-most-important-cloud-advances-of-the-decade/. Accessed: 2025-02-07. 2024.

[102] The Chromium Project. *dcsctp_options.h*. Accessed: 2024-10-06. 2023. URL: https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/net/dcsctp/public/dcsctp_options.h;.

[103] M. Thomson and C. Benfield. *HTTP/2*. RFC 9113. June 2022. DOI: 10.17487/RFC9113. URL: https://www.rfc-editor.org/info/rfc9113.

[104] C. I. Use. *Can I use - Browser Support Tables for Modern Web Technologies*. https://caniuse.com/. Accessed: 2024-08-09. 2024.

[105] W3C. *QUIC API for Peer-to-peer Connections*. https://w3c.github.io/p2p-webtransport/. Accessed: 2024-10-06. 2024.

[106] W3C. *WebRTC 1.0: Real-time Communication Between Browsers*. https://www.w3.org/TR/2011/WD-webrtc-20111027/. Accessed: 2024-06-10. 2011.

[107] W3C. *WebTransport over QUIC API*. https://w3c.github.io/p2p-webtransport/. Accessed: 2025-03-02. 2025.

[108] J. Weil, V. Kuarsingh, C. Donley, et al. *IANA-Reserved IPv4 Prefix for Shared Address Space*. RFC 6598. Apr. 2012. DOI: 10.17487/RFC6598. URL: https://www.rfc-editor.org/info/rfc6598.

[109] A. Willett. *Data at ngrok: A Primer*. https://ngrok.com/blog-post/data-at-ngrok. Accessed: 2024-10-06. 2024.

[110] XDA Developers. *Reasons Self-Hosted Software is Gaining Popularity*. https://www.xda-developers.com/reasons-self-hosted-software-gaining-popularity. Accessed: 2025-02-07. 2024.