AN ACCESSIBLE WEB-BASED PEER-TO-PEER REMOTE DESKTOP CONTROL AND ACCESS TOOL UTILIZING WebRTC

By

Heston Friedland

Dr. Mengjun Xie
Professor of Computer Science and Engineering
(Chair)

Dr. Dalei Wu
Associate Professor of Computer Science and Engineering
(Committee Member)

Dr. Yu Liang
Professor of Computer Science and Engineering
(Committee Member)

AN ACCESSIBLE WEB-BASED PEER-TO-PEER REMOTE DESKTOP CONTROL AND ACCESS TOOL UTILIZING WebRTC

By

Heston Friedland

A Thesis Submitted to the Faculty of the University of
Tennessee at Chattanooga in Partial
Fulfillment of the Requirements of the Degree
of the Master of Science

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

May 2024

ABSTRACT

In the modern tech environment, remote desktop sharing is very popular and often much-needed for daily work. Yet, many existing solutions hinge on the conventional client-server model, necessitating additional tools and software for effective desktop access. There exists a notable research gap concerning Desktop-as-a-Service (DaaS) delivery via a peer-to-peer architecture. This study introduces a browser-centric web application leveraging peer-to-peer communication for seamless remote desktop access. By integrating state-of-the-art technologies including Google’s WebRTC framework, STUN servers, and signaling servers, we offer an in-browser remote desktop experience, evaluating its performance in terms of responsiveness and user-friendliness. Our findings indicate promising prospects for WebRTC-driven remote desktop platforms.

DEDICATION

I would like to dedicate this thesis to my beloved partner Emery, who has provided so much emotional support during my time at graduate school. I am eternally grateful to her for allowing me to see this process through to the end and for always being there for me.

ACKNOWLEDGEMENTS

First and foremost, I would like to immensely thank Dr. Mengjun Xie for teaching me and taking me on as a Graduate Assistant early in my master’s degree, allowing me to get immense practical experience in multiple disciplines within the field of Computer Science, as well as providing encouraging support throughout this process. I would also like to thank my fellow Nexus Lab members, in particular Ruipeng Zhang and Chen Xu for helping me decide on my thesis topic and guiding me through my early years of graduate school and teaching me so many new important skills. Furthermore, I would like to thank Dr. Yu Liang and Dr. Dalei Wu for agreeing to be on my committee and providing their assistance. Finally, I would like to thank fellow graduate student Ayman Gumaa with whose support I was finally able to create my final application.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API, Application Programming Interface

AWS, Amazon Web Services

CPU, Central Processing Unit

CSS, Cascading Style Sheets

DaaS, Desktop-as-a-Service

GUI, Graphical User Interface

HTML, Hypertext Markup Language

I/O, Input/Output

ICE, Interactive Connectivity Establishment

IP, Internet Protocol

IT, Information Technology

MJPEGS, Motion Joint Photographic Experts Group

MMT, MPEG Media Transport

NAT, Network Address Translation

P2P, Peer-to-Peer

RDP, Remote Desktop Protocol

RFB, Remote framebuffer

ROT, Remote Order Transmission

RTC, Real-Time Communication

RTT, Round-Trip Time

SDP, Session Description Protocol

SSL, Secure Socket Layer

STUN, Session Traversal Utilities for NAT

TURN, Traversal Using Relays around NAT

UDP, User Datagram Protocol

VDC, Virtual Desktop Clouds

VDI, Virtual Desktop Infrastructure

VNC, Virtual Network Computing

VPN – Virtual Private Network

WebRTC – Web Real-Time Communication

CHAPTER 1

INTRODUCTION

Currently, in the field of computing, there is a growing need for access to machine resources – and it is growing exceedingly necessary that large organizations and typical users have an accessible method for acquiring such resources. As a result, virtual desktop technologies such as Virtual Desktop Infrastructure (VDI), Virtual Desktop Clouds (VDC), and Desktop-as-a-Service (DaaS) have become standard services that are widely offered across many platforms. They provide a method of virtualization for end users to gain access to a wide variety of machine types that may otherwise be unavailable. Furthermore, these technologies are key in providing not just local virtual machine systems, but access to remote desktop machines as well [1].

Thus, to combine many of the current tools available in the remote desktop field and provide a solution to many of the current issues that both users and administrators face, a peer-to-peer architecture for connections of remote desktops, P2P-Connect, is proposed. Through the utilization of the web-based screen capture technology through the browser-based JavaScript framework Electron, P2P-Connect aims to make the remote desktop generally more accessible than other methods which, for example, may require installation of a separate VNC viewer application on the user's end to access the virtual machine running the VNC server. Additionally, the implementation of the WebRTC protocol provides a simple API that is open source, easy to modify and customize, and aims to increase or at least match speed when compared to traditional technologies and software such as pure WebSocket communications and noVNC-based implementations of remote desktop protocols.

Background

Web Real-Time Communication

Published as an open-source technology in 2011, Web Real-Time Communication (WebRTC) is a relatively novel web-based protocol that allows for a browser or mobile application to interact with a variety of data and media sources in real time. WebRTC is supported by most of the major browsers, including Google Chrome, Mozilla Firefox, and Opera. Furthermore, WebRTC has components for becoming accessible via built-in Web APIs that are easily interactable using a variety of programming languages [2], [3]. Several components go into the WebRTC connection establishment process, including STUN/TURN servers, ICE candidates, SDP information, and signaling servers, which all result in a fast UDP (User Datagram Protocol) based peer-to-peer connection over the web. Figure 1.1 shows a basic WebRTC communication pipeline using signaling servers to establish an RTC Peer Connection to exchange media data.

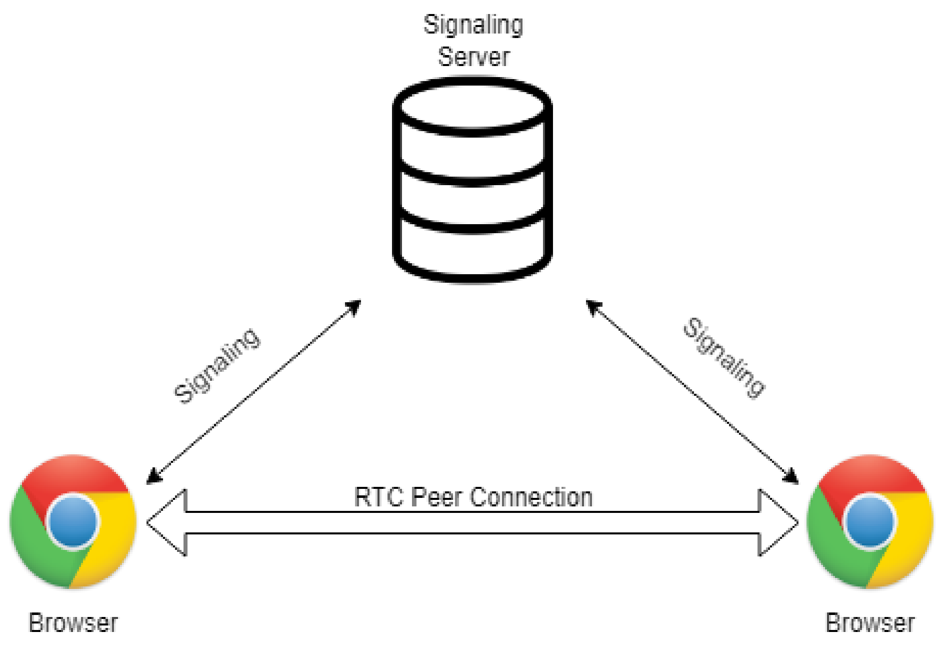

Figure 1.1

Basic WebRTC Communication Pipeline

ICE, STUN, and TURN

One of the most prevalent issues in the design of the peer-to-peer communication scheme is allowing the peers to talk to one another, despite peers typically residing within a private subnet, rather than having a public IP address that is visible to peers outside the network. As a result, it is necessary to bypass these NAT (Network Address Translation) restrictions via NAT traversal. Thus, WebRTC leverages the ICE (Interactive Connectivity Establishment) protocol, which is an established method in which peers can generate traversal candidates that can tell the other peers how to communicate with one another. The two most popular techniques to implement the ICE protocol are in the form of either STUN (Simple Traversal of UDP over NATs) or TURN (Traversal Using Relay NAT) servers as stated by [4]. Figure 1.2 shows the typical event order of requests between two peers on a STUN server. Initially, binding requests are sent, then binding responses are sent back and determine if NAT is being used. This allows the peers to then communicate their public IP and port number to one another to allow them to connect behind NAT.

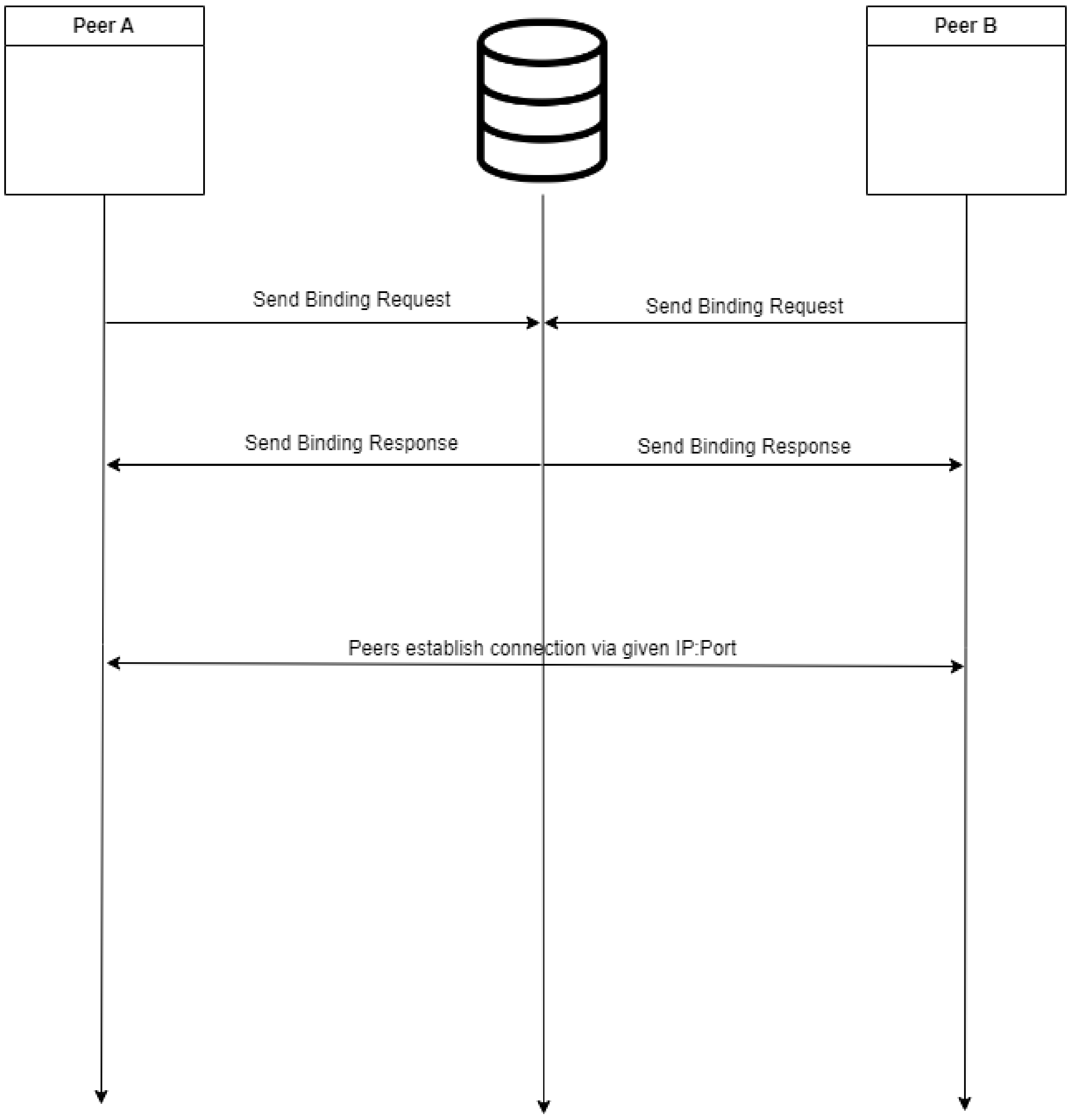

Figure 1.2

STUN Events

A STUN server allows clients to send their respective IP and port combinations that they are using to communicate and in the same way, acquire that information about other clients from the STUN server so that the client knows with what machines it is able to communicate. However, this exchange of information is only possible if the machines that are trying to communicate with one another are all behind normal or restricted NAT types. In the case that a client is behind symmetric NAT, this style of communication will fail, and a TURN server must be used instead. The TURN server acts as a traditional client-server mechanism in order to facilitate communication between two peers, with the TURN server acting as a relay for the peers.

Signaling

The final main communication protocol required to enable WebRTC is a signaling protocol. It is a necessity that for two or more RTC peers to establish a connection via the ICE Candidates via a STUN or TURN server, they must first signal to each other their Session Description Protocols (SDPs), which contains all of the multimedia information that the two peers will use in their established RTC peer connection. An important aspect of signaling is that it can essentially be done via any method that allows information to get from one peer to another. One of the most common methods of signaling is to utilize WebSocket connections. These socketed connections allow for two-way communication over a single TCP socket via a web client and a remote host, allowing users to instantiate the necessary SDP information within the RTC configurations before establishing the UDP-based RTC peer connection.

Contributions of Study

The following are the contributions of this study:

- A remote desktop accessibility application using WebRTC, which has little exploration for use in the field of Desktop-as-a-Service.
- A peer-to-peer architecture for a distribution service of remote desktops. Whereas most current solutions provide a client-server model, this paper explores a peer-to-peer architecture as a promising potential model for alleviating the issue of server overhead. This is particularly important for smaller organizations that cannot support full integration of multiple servers and the load balancing overhead that comes along with those servers.
- Finally, a way to traverse typical NAT restrictions without requiring the use of additional tools such as a reverse proxy server to expose machines behind a private NAT to a publicly facing application. Additionally, by reducing the amount of persistent middle points within a network infrastructure, less provisioning will ideally be required for service managers and administrators when maintaining services within an organization.

Organization

The remainder of the paper is organized as follows. Chapter II examines the related works on remote desktop services and technologies. Chapter III provides an overview of the methodology and tools used to develop the architecture of the solution. Chapter IV will provide the findings of the design analysis. Finally, Chapter V discusses the results and future developmental possibilities, and a short conclusion is given.

CHAPTER 2

LITERATURE REVIEW

In this chapter, thorough descriptions and reviews of the literature relating to virtual machines and virtual desktop infrastructures are given. Specifically, this chapter examines the different models that are used to serve virtual desktops for both viewing as well as control. It also details some key accessibility tools used to improve current virtual desktop solutions. Section I details several architectures that use a client-server model for desktop distribution. Section II examines the architectures that use peer-to-peer connections for distribution. Section III reviews other architectures focused on accessibility and security as primary components for distribution. Finally, section IV discusses and summarizes the chapter.

Client-Server Model

The most common form of distributing virtual desktops is through a client-server model, wherein a client accesses the virtual services provided by a server node on the network. This model provides an easy way for providers to grant access to a virtual environment for the end user.

Kim *et al.* in [5] explored the resource utilization of cloud-based virtual desktop infrastructure, particularly the increased bandwidth usage of the virtual desktop display protocols. They describe that while many proprietary virtual desktop display protocols are improving in terms of the user experience via the implementation of feature enrichment such as audio and video services on the remote desktop, these protocols are continuously increasing in bandwidth usage. As a result, they propose a new design and implementation of the virtual desktop system using a lightweight virtual desktop display protocol based on cloud Desktop-as-a-Service (DaaS). It was concluded that the most efficient method of reducing the bandwidth of the display protocol was to reduce the size of the display session data using the MJPEGS video compression algorithm.

Motivated by the increasing demand for the use of IT resources, and the issues that this increasing demand poses, [6] investigates remote desktop virtualization technologies and thus proposes a new desktop virtualization system, FastDesk. The research claims that FastDesk provides a robust solution for improving upon many popular remote desktop virtualization platforms, which are purported to be lacking in performance in terms of response time, quality, and cost. FastDesk works utilizing a server-push mechanism, which is operated by one or many management nodes that serve as the centralized control nodes that control the service nodes and manage their resources. These service nodes are the physical machines in which the virtual machines are run, depending on server requests. Utilizing a variety of algorithms and improved video streaming technology, FastDesk can achieve low CPU utilization, quick response times, and efficient bandwidth utilization.

While desktop virtualization has been greatly explored, much of the research and technology developed for remote desktop virtualization is centered around providing an entire desktop experience, reducing latency for the user, and user experience. However, Lai *et al.* claim in [7] that there has been a low focus on the granularity of the system, as well as protocol optimization. Thus, they propose a lightweight desktop virtualization system. This system allows for single application windows to be shared, as well as new protocol development to control the interactions between the clients and servers. The main tools used to create this system are application streaming to reduce application costs while increasing computing resource utilization, as well as the Remote Order Transmission (ROT) protocol, which is a platform-independent client-server protocol that reduces the data processing overhead and increases portability among the systems.

While traditional remote desktop services provide utility in the form of resource consolidation, reduced client overhead, and ease of management, many of the currently available tools only allow for low-motion graphical cases, such that video streaming or real-time interactivity becomes an issue. Thus, the streaming based remote interactivity architecture - SRIDesk - is proposed in [8]. In order to achieve its high playback rate, SRIDesk implements a server-push streaming mechanism and low-bandwidth high-performance encoders, such as the H.264 encoder, to allow for low-latency client-server synchronization of the display. Compared to other remote desktop services, SRIDesk is comparable to or better than many other tools regarding interaction response times, video quality, scalability, and resource consumption.

The client-server model has several scalable remote desktop services that not only allow for desktop sharing but full desktop control as well. However, this requires both management of the client machine as well as the distributing server to ensure the end-user experience. This model is contrasted by the relatively more easily managed peer-to-peer model, which uses a distributed network of nodes to access and supply virtual desktop resources.

Peer-to-Peer

Several tools were developed for desktop sharing that utilize peer-to-peer network architectures. These architectures mainly rely on a WebRTC-based approach to achieve their peer-to-peer configurations. For instance, Iwata *et al.* in [9] detail a solution to allow for greater usability of a local desktop through the implementation of an any-window sharing mechanism on remote desktops, such that the remote desktop takes up much less display utilization of the local desktop. This has many use cases, particularly in collaborative work efforts where one wishes to share a particular window of an application for shared editing or viewing, without wanting to share access to the entire desktop. This architecture utilizes a smartphone to control the windows to be accessed on the different machines. Wi-Fi is used to enable the sharing of files and application window functions between the systems, and Bluetooth is used to connect the smartphone and machines with a single operation device. Furthermore, the file-sharing and application window-sharing mechanism uses the WebRTC API to leverage these utilities. The host PC acts as a server which allows the remote clients to get the host client's peer ID for peer-to-peer connections.

With a focus on the increased importance of online collaborative tools, Lucas *et al.* propose a WebRTC-based solution in [10] for synchronous multi-user collaboration, known as "USE Together". This solution is unique in that it provides multiple improvements over many of its modern counterparts, including a low-latency user experience, ease of access, encrypted streams for heightened data security, and flexibility of deployment. This is accomplished by implementing a specific architecture that enables multiple users to connect to each other through a signaling server. First, the host user who initiates the connection activates the USE engine, which is composed of two parts, namely the USE Engine Core and USE Engine GUI, which handle data transmission and GUI features respectively. Once the USE engine is activated by the host, the signaling server can then handle the Peer-to-Peer communication between the host and remote users, allowing the remote users to easily join the host session and utilize the collaborative intent of the tool.

While these tools are successfully able to leverage the peer-to-peer nature of RTC, the virtual desktop service isn’t complete. Most of the modern literature simply provides a video stream or desktop viewing experience. There doesn’t seem to be much literature about full desktop control using peer-to-peer architectures.

Accessibility and Security

Zhang *et al.* in [11] detail a solution that solves the problem of a user not being able to maintain unique desktop configurations, even if running the applications on the same machine, allowing a user to run personalized software on any computer, as long as it is compatible, across the internet, even if the software does not exist on the local host being used to access the software. Furthermore, the software configurations and customization options can be saved for use across multiple systems. The main steps used to implement this solution are as follows. First, they address the issue of the installation of the applications that are intended to be run. This is done via a combination of an installation snapshot and API interception. The installation snapshot is created using a system-monitoring tool to log any changes made to the hard disk, registry, and configuration files during the application’s installation process. Then, the installation snapshot is made accessible to the application’s executable file via an API interception, which intercepts calls from the underlying system and reroutes the necessary calls to the private registry which holds the unique configuration data. To facilitate the downloading of the portable application, a P2P architecture is implemented using a dedicated server that uses a BitTorrent protocol in the form of libtorrent.

Accessibility is a very important concern for the user of virtualization technology. [12] addresses this issue and describes potential solutions to common accessibility issues using the noVNC service in conjunction with a remote desktop proxy. Using customizable virtual machine technology, users can save and reload their settings easily, and thus quickly begin access to their system. The main system architecture they propose integrates a Web-Socket proxy server, which sits between the VNC server and the VNC web client. The proxy server receives commands from the web application and relates these inputs to the VNC. Furthermore, the proxy server can process screen content for the web application as well. Finally, CLEVER is used as the VIM for the management of virtual machine environments.

In order to facilitate the usage of remote desktop applications, Sridhar *et al.* examine a solution that implements IP tunneling. Several algorithms are designed to implement the necessary tools required to run the remote-server application of both the client side, which the user accesses, and the server side, which hosts the server and desktop to be shared. The client-side consists of an HTML page, using JavaScript to capture any events that occur on the client's page. AJAX is then used to send requests to the server, as well as update the client image every time that the server updates in response to these events. This system can be implemented in both a local network, as well as over the internet. The local network deployment requires minimal tool implementation, using Flask to run the server, and the algorithms handle the remote desktop application. For over-the-internet deployment, IP tunneling is required to allow the local machine to be accessed from outside the local network. The ngrok service is utilized in order to facilitate this IP tunneling, where an ngrok client exposes the port to the ngrok server, creating a tunnel to the locally hosted server, which can be used to send and receive responses.

Li *et al.* propose in [12] a new protocol to facilitate mobile desktop-as-a-service systems. They implement the MPEG Media Transport (MMT) system format which acts as the media transport for streaming video in order to accommodate screen rendering in the system. In conjunction with MMT streaming, a screen content coding extension is used for the compression of the desktop screen. This architecture also utilizes the Google Congest Control algorithm to maintain an efficient sending rate based on network probing in order to prevent congestion issues within the network. These tools allow for a very simplified DaaS infrastructure and reduced bandwidth consumption while maintaining a similar visual quality when compared to the Red Hat SPICE DaaS application.

[13] examines improved security for the Windows Remote Desktop Protocol (RDP). They detail that while RDP has many security features including user authentication and encryption, the network is still vulnerable to attacks. The two main attacks of concern are password guessing attacks, wherein the username and password credentials of the user are at risk of being guessed. The other is a man-in-the-middle attack, wherein the attacker can impersonate the real server and steal credentials using this method. Thus, they propose implementing the RDP over an SSL VPN, which allows for credential authentication through digital certificates, as well as establishing a monitored connection from the VPN to the application server to ensure validity when the server information is relayed to the client.

In [14] a new device, ProGun, is proposed in order to address security issues that arise as a result of remote access to systems. ProGun works as a USB dongle hardware device that has been hardened and restricted to allow for secure access from a local machine to a remote location. A VPN configuration sits inside the USB, which fixes the connection from the local machine automatically and forwards all connections through the VPN tunnel. Furthermore, the USB has logging features to keep track of which device malicious network activity is coming from for improved security. Additional accessibility features are also implemented in the ProGun, as user data can be stored and reloaded whenever needed from the ProGun, e.g., the remote desktop protocol settings for the user.

Summary

Several different aspects of the literature were examined regarding the development and usability of Desktop-as-a-Service structures. Both client-server models and peer-to-peer models were examined in terms of their performance and design philosophy. Among the results that do exist, most are bulky, requiring provisioning and installation from a user to either run the tool or access the remote desktop. Additional parameters related to remote desktop services were also examined, such as research involving the accessibility of a remote desktop, as well as the security of remote desktops. It was shown that relatively little peer-to-peer data exists in the field of remote desktop control, and even less utilizing RTC in general.

CHAPTER 3

METHODOLOGY

In this chapter, a description of the methods used to create and serve the virtual machines and necessary networking components. The primary motivations for this design choice are given in Section I. Next, in section II, an overview of the architecture and justification for the various tools and services chosen is given. In addition, an in-depth examination of the various communication endpoints is given in section II. Then, the specific implementation of the primary methods and libraries used is given for both the front-end and back-end of the project in section III.

Motivations for Design

The P2P-Connect platform is designed to provide an easily accessible and manageable remote desktop environment. It aims to give remote desktop distributors a way to manageably provide a remote desktop environment to users via its peer-to-peer network setup, reducing the load on the servers handling the distribution, while keeping maintenance and management overhead to a minimum, a problem commonly seen in client-server models. Additionally, the various communication protocols can allow for a simple way to bypass NAT restrictions on both ends of the architecture, allowing private machines to be accessed from a public web application. Furthermore, it aims to allow users to access the remote desktop through a simple web application interface, where they can connect to the provided machines with nominal knowledge and application setup.

Architecture

The system design of P2P-Connect is targeted around allowing a private machine that is not exposed to the internet, such as one behind a restricted NAT, to easily communicate with a web server that serves the remote desktop via the browser. In total, there will be at least three different communication endpoints during the set-up phase of the connection, or more depending on how the administrator decides to establish the RTC connection with the remote desktop. Each of these endpoints serves a unique purpose in the P2P-Connect networking schema, and as such are necessary components for the system to run effectively. A detailed full-scale application architecture diagram is given in Figure 3.1.

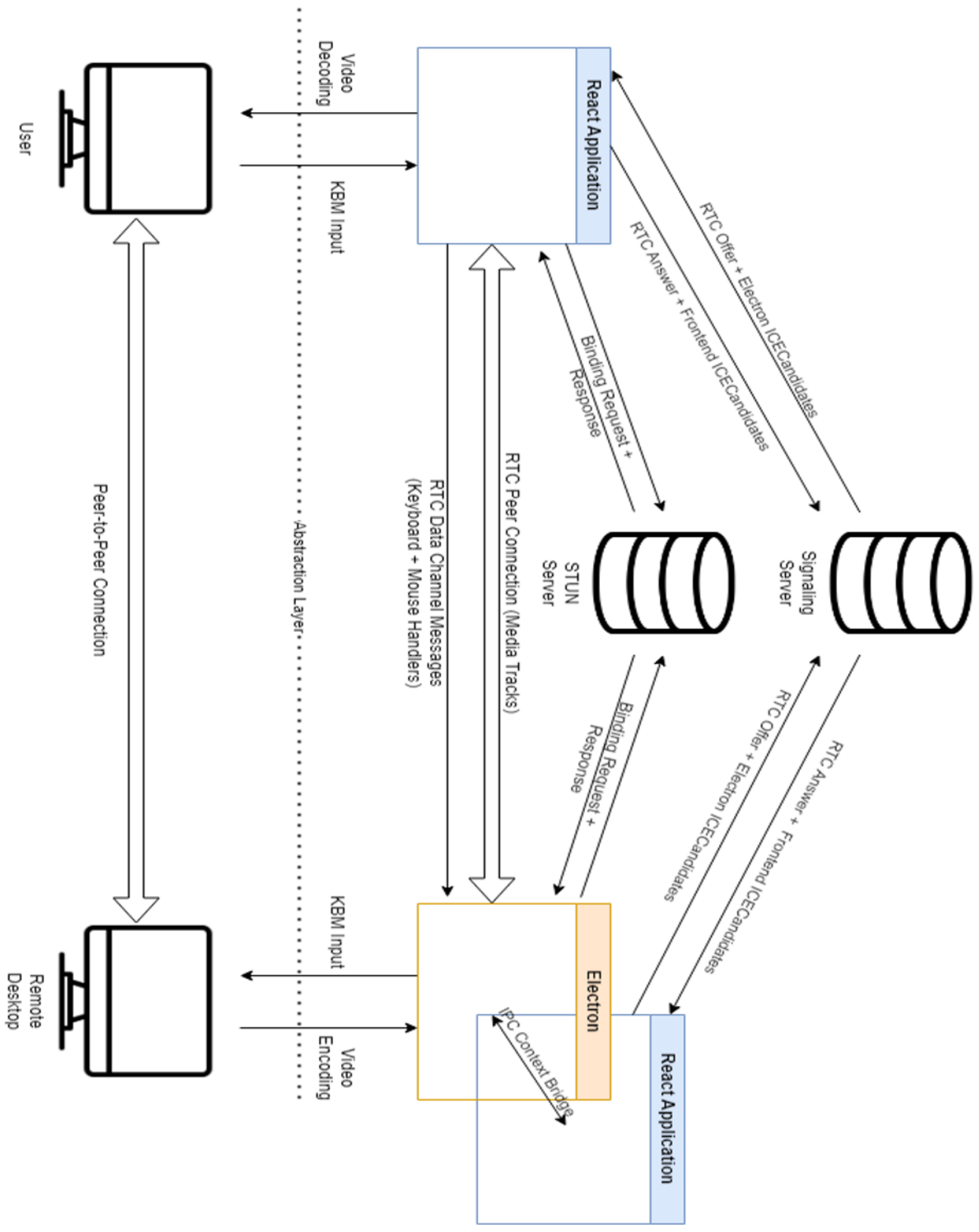
React Application
User
Video Decoding
KBM Input
Abstraction Layer
Peer-to-Peer Connection
RTC Data Channel Messages (Keyboard + Mouse Handlers)
RTC Peer Connection (Media Tracks)
Binding Request + Response
STUN Server
Signaling Server
RTC Offer + Electron ICECandidates
RTC Answer + Frontend ICECandidates
Binding Request + Response
RTC Offer + Electron ICECandidates
RTC Answer + Frontend ICECandidates
Remote Desktop
KBM Input
Video Encoding
Electron
IPC Context Bridge
React Application

Figure 3.1

Full-scale Application Architecture

Remote Desktop Endpoint

The first critical endpoint is the remote desktop that is to be shared with the public webserver. It is on this remote desktop that almost all of the libraries and tools necessary for the function of the system reside. These libraries include a JavaScript-based desktop application framework known as Electron and a React framework application back-end to instantiate the WebRTC connection and transmit this data to the web server endpoint. The primary focus of the back-end logic for the remote desktop endpoint is serving the captured desktop through an RTC peer connection, rather than through a common display protocol such as the remote framebuffer protocol (RFB) that is used by VNC. This Electron desktop is the primary tool that results in the real-time streaming of the remote machine's desktop to the public webserver. This is accomplished by capturing the screen of the remote desktop, and then passing the media stream through the RTC peer connection to the other peer, where it can load and interact with the video stream via the public webpage's canvas. Additionally, there is some logic dedicated to instantiating the peer connection and sending its SDP information, which is done via a WebSocket connection using the socket.io library in combination with the signaling logic and STUN server set up using the open source coturn library.

Public Web Application Endpoint

The web application is created using a combination of HTML, CSS, JavaScript, and React in order to serve the public-facing webpage. This webpage exists to render and stream the captured desktop media stream data that is transmitted over the open RTC peer connection in the browser. It is in this browser that the remote-control logic is created using the canvas elements within the React application renderer. Using the RTC data channel created in the remote desktop endpoint, which is an additional bidirectional arbitrary data transference channel separate from the one used to stream media over the RTC peer connection while remaining attached to the peer connection, the React app can send real-time events that capture mouse and keyboard events happening within the canvas of the public React application. This simultaneously provides user accessibility to the remote desktop, while keeping the rest of the private network relatively obfuscated from the users, only giving them access to the hosted web application and the single remote desktop, which may reside on different networks, allowing for more fine-grained control of the remote desktop's security.

STUN and Signaling Endpoints

The remaining necessary endpoints involved in the establishment of the connection among the peers are solely dedicated to transmitting the information sent from one peer to another or facilitating that transmission. The STUN server is the key component for allowing the remote desktop and web server to communicate with one another, even behind NAT. The STUN server is created using the open-source library "coturn," due to its ease of setup and reliability. While there are several public STUN servers available to use, creating one's own server allows for more flexible management of the STUN configurations, and is thus much more maintainable, and prevents any errors from a sudden third-party shutdown or denial of service such as during server maintenance.

The signaling server logic is contained in both the public web application as well as the back-end react logic that waits for specific SDP types to be broadcast to the server hosting the socket.io server, namely the offer and answer from the peers, collects the messages that contain these types, and then relays the SDP information from one peer to the other depending on whether the type was an offer or an answer. The scripts utilize socket.io, a JavaScript wrapper for WebSocket connections chosen for its reliability and ease of use, to transmit these descriptions to and from each peer which allows them to set their local and remote session descriptions and transmit ICE candidates to establish the connection that instantiates the media and data channels between the peers.

Implementation

A combination of both front-end and back-end tools is necessary for the creation and implementation of the architecture. The main approach utilized for the peer-to-peer architecture involved using a combination of the WebRTC protocol and JavaScript/React for creating and interfacing with the front-end web application, and Electron, a JavaScript-based framework to capture the desktop and transfer the remote desktop’s screen information using the media stream to the public webserver.

Front-end Implementation

Sitting on the front-end of the public web server is a React application created using npx, a JavaScript/Node.js package execution tool. React is a component-based JavaScript application framework that allows for the easy production of quality JavaScript code, with many quality-of-life features over regular JavaScript pages. JavaScript was chosen as the base of the frameworks as it provides an intuitive and flexible platform for web interactivity that can be effortlessly read and duplicated. On the public web interface, the offer from the remote desktop application is replied to with a new RTCPeerConnection(), SDP answer, and any generated ICE candidates, which encompasses all the information required for the public webserver connection to complete its instantiation of a peer connection with other users. Among the other critical functions executed on the public webserver are the creation and capture of the mouse and keyboard events on the React webpage. Using React’s keyboard and mouse handlers, data such as the mouse location and click or key tap events can be captured within the canvas of the web application.

Furthermore, the new peer connection allows the sending of data across the RTC peer connection’s data channels, which are established via the remote desktop application using the createDataChannel() method. This channel enables arbitrary data transfer across the peer connection, crucial for facilitating user access to the remote desktop. Among the critical functions executed on the public webserver are the creation and capture of the mouse and keyboard events on the React webpage which utilizes this data channel to funnel events to the other peer. For instance, the OnMouseMove() function handler will extract the coordinates from the mouse location on the React page’s canvas. Once these coordinates are extracted, the event handler can then send these coordinates across the data channel at very fast speeds, allowing almost every single mouse event on the user’s end to be sent over to the remote desktop application where the mouse movement logic can be processed. Similar logic applies to mouse and keyboard interactions. While the remote desktop primarily manages the underlying logic processing, the front-end is responsible for capturing the necessary data over the data channel and ensuring it is transferred over the data channel without any abnormal interaction from the user. This aims to create a user-friendly interface in which accessing the remote desktop is as effortless and efficient for the end-users as possible.

Remote Desktop and Back-end Implementation

The back-end of this system is also powered by a separate React application, which sits behind an Electron server that collects all of its data to render and generate any key events passed from the public webserver. The React application sitting in the App.js file handles much of the initialization logic for setting up the RTC peer connection to the front-end peer. When the screen is shared, the back-end React server creates a media stream object that can be added to the peer connection and captures the current desktop based on the selected screen. The React app simultaneously creates the offer to the other peer using the createOffer() method and generates the necessary SDP information that is sent to the front-end web application via the signaling server, while also handling the answer returned from the front-end using the setRemoteDescription() method. Additionally, like the front-end, it also has the necessary signaling logic to handle the ICE candidates that are generated. It also handles the creation of the RTC data channel, where arbitrary data, such as the mouse movement, mouse clicks, and keyboard tap events, are able to be related from one peer to another, i.e., from the front-end peer to the back-end React server to be rendered via the Electron application.

In addition to the React application, a Node.js framework, Electron, is also utilized to create the interface with which the back-end logic is rendered and selected. As it is a Node-based application, it is able to be controlled as well as interact with the remote desktop in an automated manner. Electron uses a context bridge to ensure that any data that is passed to the React application can be rendered in the Electron application. Thus, the data that is passed to the back-end server is also able to be captured by the Electron application, which can then be used to send mouse and keyboard events via an automation library, Robotjs. As data is sent from the front-end to the back-end React server, Electron is able to utilize these data channel messages and automatically control the remote desktop. Figure 3.2 details the typical flow of the full event capture from the front-end to the Electron application, including the data channel and context bridge intermediaries.

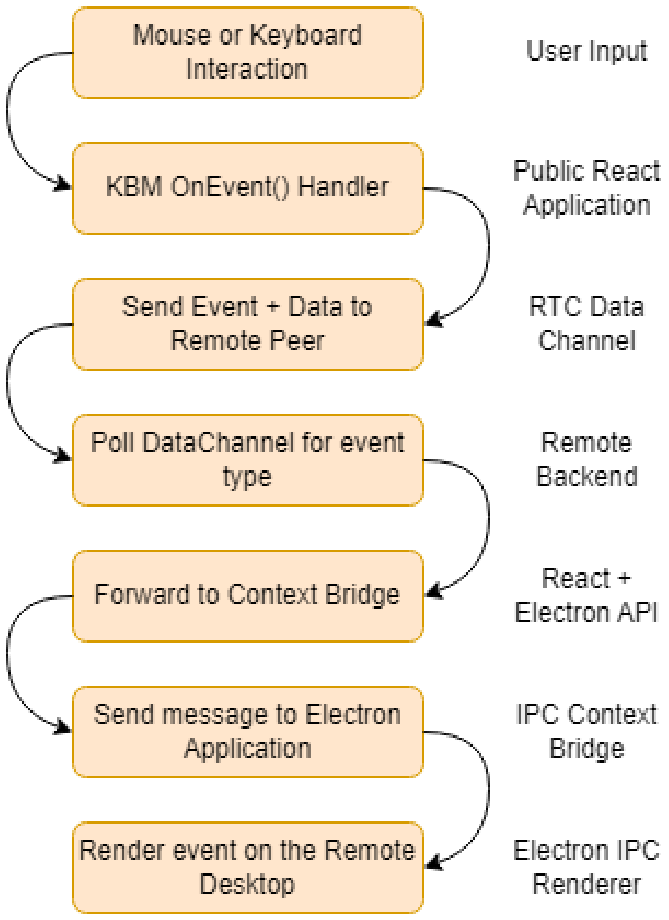
Mouse or Keyboard Interaction
User Input
KBM OnEvent() Handler
Public React Application
Send Event + Data to Remote Peer
RTC Data Channel
Poll DataChannel for event type
Remote Backend
Forward to Context Bridge
React + Electron API
Send message to Electron Application
IPC Context Bridge
Render event on the Remote Desktop
Electron IPC Renderer

Figure 3.2

Order for Handling Mouse and Keyboard Events

Signaling Implementation

The signaling logic is a basic WebSocket server implementation utilizing the socket.io library. Given certain events within any peer connected to the socket.io server host, such as messages with an ‘offer’ or ‘answer’, or ‘icecandidate’ type, the WebSocket server is able to broadcast the messages with any associated data to the other peers connected to the socket server via the established TCP connection. Thus, when either peer receives these messages, they can reply to the WebSocket server which can then broadcast the reply to the other peers, allowing a peer-to-peer connection to be established. At this point, the signaling logic is not necessary to continue communications, however, it remains best practice to keep the connections open if the ICE candidates update or there is an interruption in the current peer connection that needs to be re-signaled.

CHAPTER 4

RESULTS

In this chapter, an examination of the results of the designed architecture of a remote desktop-sharing application will be conducted. Section I will discuss the evaluation parameters used to analyze the application. Then, Section II will include the results of the analysis of the metrics for the application, as well as a comparison against a few other commonly used remote desktop-sharing applications in terms of both quantitative performance as well as quality-of-life/accessibility parameters.

Evaluation Parameters

In order to examine the efficacy of the peer-to-peer architecture, different scenarios of load are tested across several platforms using several common metrics in remote desktop applications. The latency of the signaling server connection, the Round-Trip Time (RTT) between the peer connections used to transmit the video data, as well as the latency for basic keyboard and mouse events that are sent over the data channel within the RTC peer connection are all measured. Other additional metrics such as the frames per second of the encoded shared video, packet loss, and frames dropped will also be provided for the WebRTC-based connection. These tests will be performed using a combination of machines including Linux and Windows-based user endpoints to view a remote host while using Google Chrome as the web browser due to the current functionality of WebRTC implementations in other browser selections. Furthermore, these various factors will be checked under different load uses, such as web browsing, playing a video, and keyboard/mouse interaction to collect a wider array of data points.

After such tests are completed, they are compared against the available results from a similar architecture utilizing vncserver with noVNC together sitting behind a reverse proxy to emulate a comparable system that uses NAT traversal techniques akin to the WebRTC architecture. The noVNC and VNC protocols were chosen due to their widespread usage in remote desktop applications, as well as the similarity in design philosophy utilizing only a browser for the user to access remote desktops as well as the relatively straightforward integration with reverse proxy servers. Similar parameters will be captured in these tests, at which point they will be compared to one another for a careful analysis of results. Figure 4.1 shows the basic noVNC architecture that is being used for comparison.

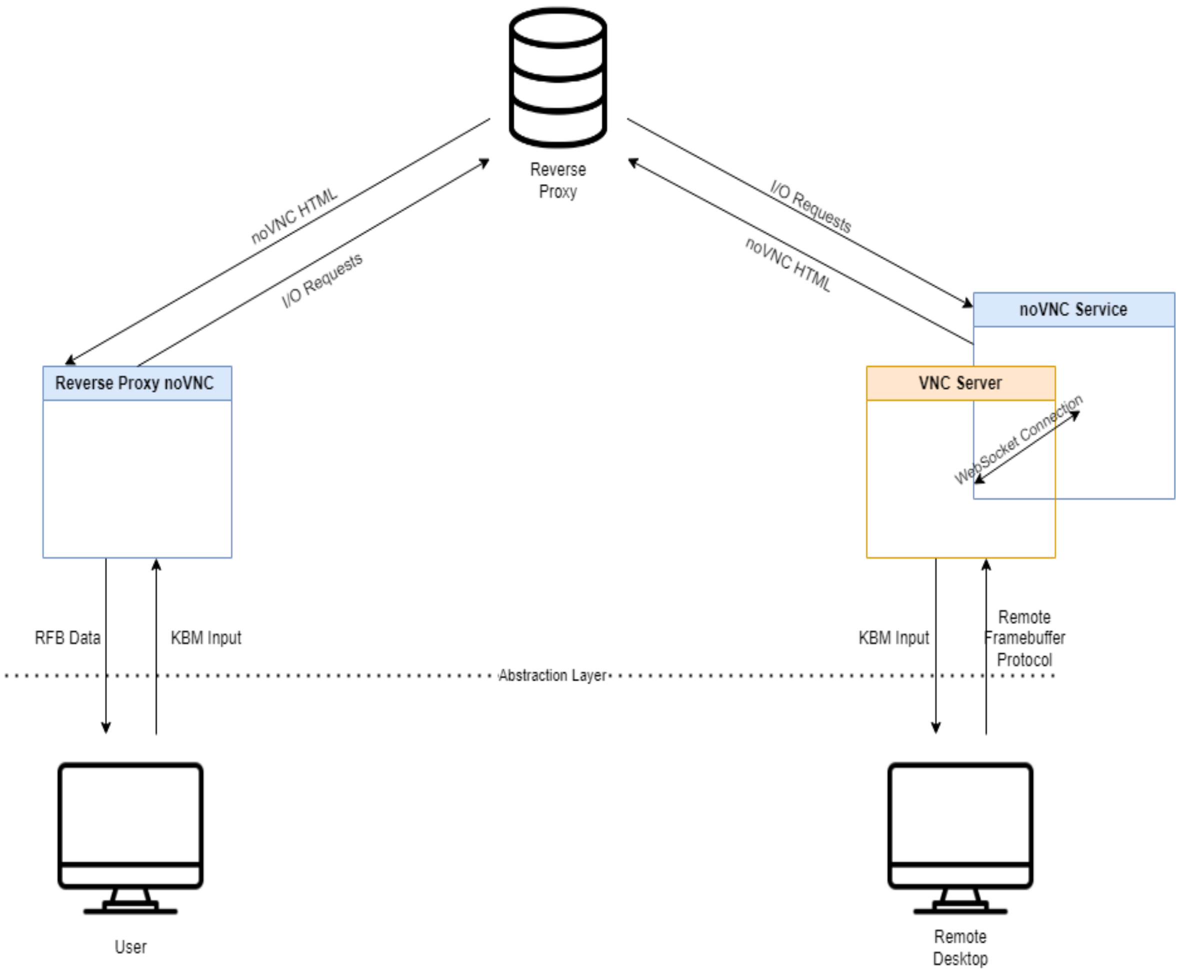

Figure 4.1

noVNC and Reverse Proxy Architecture

Network Performance Results

One important aspect to consider in deciding the efficacy of the WebRTC-based architecture is its network performance. Several metrics such as the round-trip time, connection establishments, and signaling latency will be examined for both the client and host side of the application. The screen-capture side of the application running Electron on the remote Windows 10 machine can successfully send the video component via the WebRTC peer connection video encoding, as well as handle the mouse and keyboard events that are sent from the RTC data channel. Using the built-in Date functions within JavaScript, the RTT for the socket.io WebSocket-based signaling server is ~140 milliseconds on average when sending and returning from Peer A to the Amazon Web Services (AWS) Signaling server to Peer B and back with a range of 135 milliseconds to 150 milliseconds. Using the internal Chrome RTC stats API, an RTT of 1-3 ms was found in the RTC peer connection. Then, in examining the RTC connection between the two peers after signaling, the initial delay for establishing and sending data such as mouse and keyboard events across the RTC data channel appears to be around 200 milliseconds, but once established it is able to log every single mouse and keyboard event within 1-10 milliseconds of one another, and it can occasionally send over 2 messages per millisecond. Additionally, the CPU utilization on the machine sharing the remote desktop screen is approximately 15 – 20% while active via browsing the web and other activities such as typing, and a CPU utilization of around 3-5% when inactive, meaning that no activities are being performed via the public web interface. Furthermore, data provided from Google Chrome’s WebRTC internals shows a constant 28-30 frames per second (FPS) during active use of the screen-capture application from the user’s window, with a relatively low RTT of 5 milliseconds for the peer connection. However, despite the low RTT between the two peers, the encoding and decoding process for the rendering of the desktop view adds a significant processing delay, which is around another additional 30 milliseconds, which aligns with the frames per second data that the RTC internals are displaying.

The user side of the application runs on a Windows 10 machine. On the front-end web application, the CPU usage is relatively low at .5% – 1% for an inactive browser window, increasing to around 5%-11% when generating mouse and keyboard events in the browser window. The bitrate of the peer connection is 3000-5000 bits sent/s when completely idle, 7000-

9000 bits sent/s when only playing a video in the browser and no other interaction but sees a drastic increase to around 130,000 bits sent per second when actively controlling KBM controls in the browser. The bits received/s are noticeably greater as well, which aligns with the media transference from the desktop application to the web application, with ranges from 95,000 – 130,000/ 230,000 – 270,000/ and 250,000 – 290,000 with I/O events, again aligning with the bidirectional nature of the RTC architecture of the two peers. Similar networking results are seen when running the front-end server from a Kali Linux machine on the same network.

Next, a VNC architecture utilizing noVNC being exposed to the internet via a reverse proxy on the same AWS server used in the previous tests will be examined. The initial time to establish a connection between the user and the noVNC server after routing through the reverse proxy is approximately 580-800 milliseconds if all of the assets are cached in the browser, and up to 5 seconds to establish a connection if none of the assets are preloaded. However, due to the lightweight nature of noVNC, the host machine only runs at around 3-4% CPU utilization during inactive phases and only goes up to around 7% utilization from all of the related processes to serve the noVNC client even when utilizing a heavy I/O load from the keyboard and mouse. Additionally, the RTT for sending data between the noVNC client and VNC server is around 120 milliseconds, mostly due to the connection through the reverse proxy on both ends. Like the RTC connection, multiple images are rendered in a short period, taking ~3 milliseconds between each new base 64 image to be rendered within the noVNC page according to the browser networking traffic logs. Tracking I/O event data in noVNC is not natively supported through its RFB capture, so there is no direct way to measure the mouse and keyboard events for direct RTC comparisons. Table 4.1 shows the relevant information comparing the P2P-Connnect and noVNC applications.

Table 4.1

Application Performance Comparison

	P2P-Connect	noVNC
RTT (ms)	1 - 3	~120
CPU Utilization Idle (%)	3 - 5	3-4
CPU Utilization Active (%)	15 - 20	6 - 8
Instantiation Time (ms)	200-400	580 - 800
ms/Message	0-10	3

Accessibility Comparison

Another important factor to examine outside of the quantitative network performance is the accessibility and usability of the tool. Currently, installing the VNC server, the noVNC library, along with a reverse proxy tool on a public server is quite easy to automate. On the contrary, the current solution for the WebRTC architecture requires manual selection of the screen one wishes to share, as browsers such as Chrome and Firefox have specifically placed restrictions on certain APIs within the WebRTC library that make it very difficult if not impossible to automate the selection of the desktop capture without exposing the remote desktop to great security risk. However, in terms of the user accessing the remote desktop after the connection has been established between the browser and remote host, there is relatively little difference between a reverse proxy noVNC webserver and the WebRTC web application. Furthermore, the trade-off for WebRTC's greater bandwidth usage is that the application can maintain a constant framerate, even when streaming video while the user is not active, as the noVNC/VNC protocol works to maintain bandwidth when idle, as compared to maintaining a constant latency. This is reliant on the importance of the UDP protocol, as constantly sending packets, even if they arrive slightly out of order or get dropped, allows for a constant consistent video quality that noVNC is not able to provide over its implementation.

Another important factor in the examination of the accessibility is the system agnosticism of each application. The WebRTC application is system agnostic, able to run on Windows and Linux with a simple installation using the given libraries already built into the application packages. In contrast, noVNC in combination with VNC is not easily implemented within Windows Operating Systems but is relatively easy to establish the architecture on Linux. In terms of browser support, ultimately both applications are fully capable of supporting modern browsers given the correct RTC browser adapters are placed into the libraries before running, due to the differences in RTC implementation between browsers. Finally, due to the nature of symmetric NAT, any time a user behind a symmetric NAT wishes to utilize P2P-Connect, they will first have to establish a TURN server and modify the current ICE server configurations in order to support running the application, which will ultimately increase the delay. This is particularly detrimental to those using mobile networks to maintain internet access, as most mobile network configurations reside behind symmetric NAT types. This will ultimately cause the P2P-Connect design to more closely resemble the client-server and reverse proxy models that the noVNC application is based on, which is antithetical to the design philosophy of the project goals.

Summary

The results of the application were examined under several different metrics, including round-trip time between communication endpoints, streaming bitrate, frames per second of video, and I/O latency. These results were then analyzed and compared against a noVNC reverse proxy model, which is a current solution with similarities to the WebRTC architecture due to its NAT traversal abilities while sharing a remote environment. The results showed clear use cases for both applications, particularly that the WebRTC application is effective when video fidelity and consistent performance are of importance and is usable in small-scale environments for sharing remote desktops without overburdening either system. In this way, P2P-Connect was able to prove itself as a concept for remote desktop sharing.

CHAPTER 5

DISCUSSION AND CONCLUSION

This chapter explores the primary objectives of the study, as well as the findings and potential other avenues for research complete Section I details the primary objectives that were sought out in this study. Section II then summarizes the results from the evaluation of the application. Then, a brief conclusion on the project is given in section III. Finally, future recommendations for areas of interest study are given in section IV.

Objectives of the Study

In the current field of Desktop-as-a-service, there is relatively little research related to peer-to-peer-based architectures for serving remote desktops. Most current peer-to-peer architectures are based either on a back-end application that utilizes software that both the user and the host must install, or instead based on a client-server model. The client-server model is an effective solution, however, in small to medium-sized organizations, server overhead can be a great cost. As a result, this project sought to use the web-based peer-to-peer framework WebRTC as a method to establish an easy-to-use application interface for remote desktop sharing while minimizing server costs. The primary objective of this study was to determine the efficacy of such a solution.

Another objective of this study was to find a secure method of sharing a desktop environment that sits behind a NAT. Typically when exposing multiple private machines to a service, a reverse proxy server must be used to expose their services to a public interface. This can also result in greater latency and overhead in remote desktop services. One of the key features of WebRTC is its utilization of the STUN/TURN protocols, which allow for NAT traversal, even between two peers behind NATs. This allows for a minimal setup method for two peers behind NAT to communicate with one another over the web.

Summary of the Findings

The primary objectives for the peer-to-peer architecture were successful. Particularly a successful remote desktop capture and control tool was able to be made leveraging the primary WebRTC protocol components, allowing peers to communicate with one another over UDP. Furthermore, the STUN protocol implementation within WebRTC addressed the primary issue of NAT traversal. However, the quantitative analysis of P2P-Connect against the reverse-proxy/noVNC infrastructure shows that it has specific use cases that may not be suitable for all implementations. Particularly, the remote desktop client CPU utilization was relatively greater than the CPU utilization found in the noVNC solution, so if the remote desktop has low processing capabilities, it may bottleneck performance. Similarly, on the public webserver, there was a slight increase in CPU utilization when using the browser. This increased CPU utilization, along with the increased network bandwidth, however, is necessary to ensure the high-quality stream of the remote desktop, which has applications in services that require a greater video fidelity than the lower bitrate solutions provide. Furthermore, the application can doubly serve as a desktop sharing mechanism, as the CPU utilization and bandwidth utilization are substantially lower when I/O is not being sent and processed over the RTC data channel/and rendered in the media stream. In terms of accessibility, the WebRTC met most of the ease-of-use standards, requiring minimal knowledge from the user to access the remote desktop. Furthermore, little application setup is required on the server end, the only major limitation to accessibility being found in the initialization of the screen-sharing mechanism, which requires manual user input due to the nature of the desktop capturer in the web-based Electron application.

Conclusions

In an effort to provide an accessible and efficient remote desktop sharing mechanism, a peer-to-peer connection architecture was developed. This was accomplished using a variety of different protocols and libraries. The primary communication protocols that exist in this architecture are the WebRTC protocol, responsible for creating the SDP information that determines the media stream behavior between the two peers, as well as exchanging arbitrary data in the form of mouse and keyboard events over its data channels. Other communication protocols include the STUN protocol that is used to establish a connection behind NAT for the two peers, as well as the signaling servers, which provide a communication endpoint for the users to allow them to establish the peer-to-peer connections. The other tools used include JavaScript and React to create the public webserver interface and render any user inputs to be sent over the data channel. Then on the remote desktop endpoint, a combination of React and Electron are used to capture and process user inputs.

This combination of tools proved to provide a successful build of a peer-to-peer remote desktop-sharing architecture. The performance of this architecture was measured on both its quantitative performance, using metrics such as latency, bitrate, frames per second, and CPU utilization, as well as its accessibility parameters. These results were also contrasted against a reverse proxy noVNC architecture to give scope to the objectives that this project was attempting to accomplish. The fulfillment of these objectives intends to provide future researchers with a path to examine similar and improved remote desktop-sharing applications.

Recommendations for Further Study

There is still a great deal of research and exploration that can be done in the field of Desktops-as-a-Service. The key limitation found in the accessibility of the P2P-Connect architecture was with the automation of the screen sharing decision that is performed by the Electron application running on the remote desktop endpoint. Thus, an examination of different desktop capturing methods that could stream over the RTC peer connection would potentially prove effective in automating the screen capture. For instance, if a command-line RTC client that could capture and stream the desktop were available, it could potentially solve a multitude of issues within the current solution such as the automation of setup during the provisioning of a new remote desktop environment. Additionally, the data flow on the remote desktop side of the application may be potentially improved, as the current solution of rendering a local React server in the Electron application, and then processing the data provides an additional layer of communication between the user and remote desktop control, which could be a communication point that could be optimized in future editions of the application in order to reduce client load.

Another area of research to explore is a combination of current DaaS solutions such as the P2P-Connect and noVNC solutions. This may prove to be valuable as a focal point to explore in future editions of the application by combining the best aspects of each different tool. This would ideally allow for the video fidelity and quick sent I/O events that WebRTC provides while maintaining a relatively lightweight nature that can be seen in VNC or X server implementations.

An additional important topic to potentially examine is the feasibility of provisioning these web-based remote desktop applications under greater load, particularly when multiple desktops are accessed from the same public server. It is important to examine large-scale applicability for multiple users utilizing the application at the same time as to determine the scalability and bottlenecks of the application.

REFERENCES

[1] E. Magaña, I. Sesma, D. Morató, and M. Izal, “Remote access protocols for Desktop-as-a-Service solutions,” *PLoS ONE*, vol. 14, no. 1, p. e0207512, Jan. 2019, doi: 10.1371/journal.pone.0207512.

[2] B. Sredojev, D. Samardzija, and D. Posarac, “WebRTC technology overview and signaling solution design and implementation,” in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia: IEEE, May 2015, pp. 1006–1009. doi: 10.1109/MIPRO.2015.7160422.

[3] R. Eskola and J. K. Nurminen, “Performance evaluation of WebRTC data channels,” in *2015 IEEE Symposium on Computers and Communication (ISCC)*, Larnaca: IEEE, Jul. 2015, pp. 676–680. doi: 10.1109/ISCC.2015.7884873.

[4] G. Li, Y. Ding, B. Xu, and X. Li, “Development and Research Based on WebRTC Mobile Phone Video Communication,” in *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, Chengdu, China: IEEE, Mar. 2019, pp. 2487–2490. doi: 10.1109/ITNEC.2019.8729024.

[5] S. Kim, J. Choi, S. Kim, and H. Kim, “Cloud-based virtual desktop service using lightweight network display protocol,” in *2016 International Conference on Information Networking (ICOIN)*, Kota Kinabalu, Malaysia: IEEE, Jan. 2016, pp. 244–248. doi: 10.1109/ICOIN.2016.7427070.

[6] T. Song *et al.*, “FastDesk: A remote desktop virtualization system for multi-tenant,” *Future Generation Computer Systems*, vol. 81, pp. 478–491, Apr. 2018, doi: 10.1016/j.future.2017.07.001.

[7] G. Lai, H. Song, and X. Lin, “A Service Based Lightweight Desktop Virtualization System,” in *2010 International Conference on Service Sciences*, Hangzhou, China: IEEE, 2010, pp. 277–282. doi: 10.1109/ICSS.2010.44.

[8] Jiewei Wu, Jiajun Wang, Zhengwei Qi, and Haibing Guan, “SRIDesk: A Streaming based Remote Interactivity architecture for desktop virtualization system,” in *2013 IEEE Symposium on Computers and Communications (ISCC)*, Split, Croatia: IEEE, Jul. 2013, pp. 000281–000286. doi: 10.1109/ISCC.2013.6754960.

[9] S. Iwata, T. Ozono, and T. Shintani, “Any-Application Window Sharing Mechanism Based on WebRTC,” in *2017 6th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI)*, Hamamatsu: IEEE, Jul. 2017, pp. 808–813. doi: 10.1109/IIAI-AAI.2017.211.

[10] L. Lucas, H. Deleau, B. Battin, and J. Lehuraux, “USE Together, a WebRTC-Based Solution for Multi-user Presence Desktop,” in *Cooperative Design, Visualization, and Engineering*, vol. 10451, Y. Luo, Ed., in Lecture Notes in Computer Science, vol. 10451. , Cham: Springer International Publishing, 2017, pp. 228–235. doi: 10.1007/978-3-319-66805-5_29.

[11] Y. Zhang, X. Wang, and L. Hong, “Portable Desktop Applications Based on P2P Transportation and Virtualization,” in *22nd Large Installation System Administration Conference (LISA 08)*, San Diego, CA: USENIX Association, Nov. 2008. [Online]. Available: https://www.usenix.org/conference/lisa-08/portable-desktop-applications-based-p2p-transportation-and-virtualization

[12] D. Mulfari, A. Celesti, M. Villari, and A. Puliafito, “Using Virtualization and noVNC to Support Assistive Technology in Cloud Computing,” in *2014 IEEE 3rd Symposium on Network Cloud Computing and Applications (ncca 2014)*, Italy: IEEE, Feb. 2014, pp. 125–132. doi: 10.1109/NCCA.2014.28.

[13] Longzheng Cai, Shengsheng Yu, and Jing-li Zhou, “Research and implementation of remote desktop protocol service over SSL VPN,” in *IEEE International Conference onServices Computing, 2004. (SCC 2004). Proceedings. 2004*, Shanghai, China: IEEE, 2004, pp. 502–505. doi: 10.1109/SCC.2004.1358052.

[14] K. Bicakci, Y. Uzunay, and M. Khan, “Towards Zero Trust: The Design and Implementation of a Secure End-Point Device for Remote Working,” in *2021 International Conference on Information Security and Cryptology (ISCTURKEY)*, Ankara, Turkey: IEEE, Dec. 2021, pp. 28–33. doi: 10.1109/ISCTURKEY53027.2021.9654298.

VITA

Heston Friedland was born in San Antonio, TX, to parents John and Suzanne. He is the youngest of three children, with two older brothers. He attended Sango Elementary in Clarksville, TN, and continued to Rossview Middle and High School in Clarksville, Tennessee. In 2019, he received a Bachelor of Science degree in Chemistry from the University of Tennessee – Knoxville. In 2020, he accepted a Graduate Research Assistant position, where he has worked while continuing to pursue his Master of Science degree in Computer Science at the University of Tennessee – Chattanooga.