Design and Implementation of a Secure Peer-to-Peer Remote Access System with Minimal Server Support

Denys Skira
40628952

SOC10101 Honours Project
Interim Report

Edinburgh Napier University
School of Computing
December 2025

Abstract

This interim report documents the progress of initial research and prototype development for a secure peer-to-peer (P2P) remote access system designed for minimal server dependency. During the project research and development, I explore the viability of an alternative architecture to commercial cloud-based solutions such as TeamViewer and AnyDesk. The focus of this project is on addressing privacy erosion, censorship vulnerability and operational dependency concerns. The report contains a literature review of relevant protocols and technologies, justification of the technical stack selection, description of the current system architecture and a roadmap for future development. The prototype successfully establishes encrypted peer-to-peer connections within local networks.

Contents

List of Figures

Acknowledgements

I would like to thank my project supervisor Professor Jon Kerridge and my second marker Andreas Steyven for guidance and constructive feedback throughout the first trimester until this interim phase.

1 Introduction

Remote access software has become an invaluable tool in distributed and collaborative work. Commercial instruments such as TeamViewer, AnyDesk, and Microsoft Remote Desktop Protocol (RDP) have become the most widely used solutions, but they all rely on vendor-controlled centralised cloud architecture. This type of solution introduces fundamental architectural constraints that can cause three critical problems:

- **Privacy erosion.** Connection metadata usually contains sensitive information such as IP addresses and geolocation, as well as connection timestamps along with a long list of other types of data that could be used for profiling and de-anonymisation. Users could be subject to unilateral data harvesting that could be later used for commercial or malicious purposes.
- **Censorship vulnerability.** Centralised services present a single point of failure that authoritarian regimes can take control of. The most famous demonstration of this is China's Great Firewall, which restricts commercial Virtual Private Networks (VPNs) and remote access providers.
- **Operational dependency.** Recent incidents with service outages from cloud providers like Amazon Web Services (AWS) demonstrate that reliance on central infrastructure may result in unexpected downtime. Furthermore, vendor-controlled systems have their own policies that may allow for user account termination, effectively eliminating critical communication channels without user recourse.

For vulnerable groups of individuals like human rights activists, investigative journalists, or Non-Governmental Organizations (NGOs) operating in crisis zones, these represent an existential operational risk. Classic secure communication tools like Signal offer end-to-end encrypted messaging but lack real-time remote access capabilities. On the other hand, commercial remote access platforms provide solid functionality but are fundamentally incapable of satisfying requirements for privacy-focused architectures.

This project investigates the viability of a peer-to-peer architecture with minimised centralised server coordination to provide an alternative to cloud-based remote access without compromising usability for both non-technical and expert users.

1.1 Development Objectives

The core development objectives include:

- Research, design, implement, and evaluate a prototype that allows the establishment of encrypted connections with optional relay dependency (off by default).
- Achieve solid NAT (Network Address Translation) traversal rates in real-world complex networks utilizing commercial WebRTC solutions.
- Ensure cross-platform support for Windows (Minimum Viable Product), Linux, macOS, Android, and iOS. Where platform limitations apply, include at least a view-only mode.
- Demonstrate the integration of security features commonly used in contemporary privacy-focused products, such as metadata minimisation, forward secrecy, authenticated key exchange, and cloud-less data transit.
- Collect and evaluate performance benchmarks including latency, packet loss, frame transmission rate, and file transfer speed.

1.2 Architectural Data Flow

The initial system architecture was designed to enable peer-to-peer communication while strictly limiting the signaling server to metadata handling. As illustrated in Figure 1, the system design ensures the server never handles screen data, inputs, or files. The connection lifecycle follows the numbered sequence below:

1. **Registration (Action 1):** Both Client A (Initiator) and Client B (Responder) register with the Central Infrastructure to establish presence.
2. **Discovery and Handshake (Actions 2-3):** The Initiator performs discovery (2) and invites the Responder. The server coordinates the handshake (3) to pair the clients.
3. **NAT Traversal (Action 4):** The system attempts to negotiate a direct path between clients through NAT traversal and candidate exchange.
4. **Tunnel Establishment (Action 5):** Upon successful traversal, a secure tunnel is established directly between the peers.

5. **Secure Data Streams (Actions 6-7):** Once connected, real-time screen/input data (6) and file transfers (7) flow directly between clients via the End-to-End Encrypted green zone, bypassing the server entirely.

6. **Fallback (Action 8):** In difficult network environments where direct NAT traversal fails, an optional Relay (8) can be opted-into, handling only encrypted data packets.

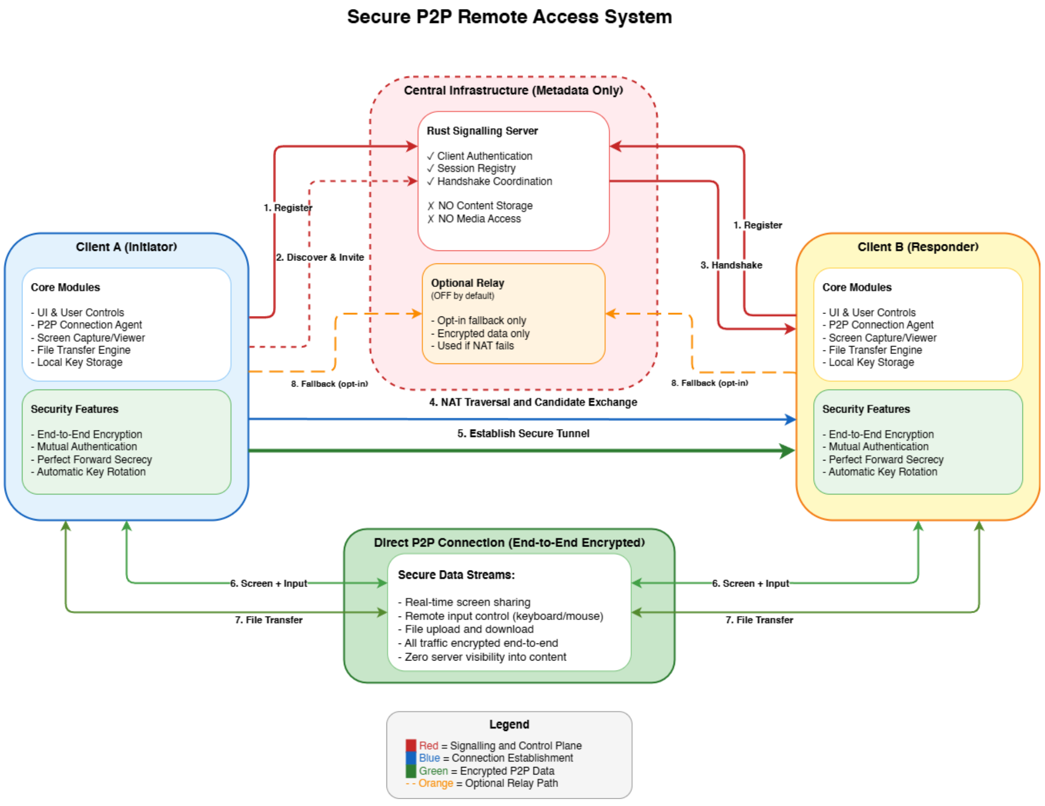

Figure 1: Peer-to-Peer Remote Access System Design Diagram V1

2 Literature and Technical Background

2.1 Remote access Protocols and Architectures

Remote access systems generally use three main architectural approaches.
Microsoft's **Remote Desktop Protocol (RDP)** [1] provides a comprehensive remote desktop experience but typically operates within trusted networks, assuming infrastructure security unsuitable for ad-hoc peer-to-peer scenarios.
Virtual Network Computing (VNC), defined in RFC 6143 [2], focuses on platform independence and simplicity, but lacks built-in NAT traversal and dynamic peer discovery.
WebRTC [3] offers the most relevant functionality because it was originally designed for internet-based peer-to-peer communication. Its architecture matches the project requirements well, offering built-in NAT traversal and using the central signalling server only to establish the initial connection parameters (SDP) as defined in RFC 8866 [4], while media flows directly between peers or via Traversal Using Relays around NAT (TURN) relays when necessary.

2.2 NAT traversal and connectivity

NAT [5] constitutes the major technical bottleneck for peer-to-peer systems. According to research, symmetric and carrier-grade NATs affect a significant portion of internet users, more commonly in mobile networks where direct peer-to-peer connectivity is often blocked [6]. The Interactive Connectivity Establishment (ICE) framework (RFC 8445) [7] provides NAT traversal through prioritised candidate gathering and connectivity verification. ICE collects candidates from multiple sources: direct address, server-reflexive candidates and relay candidates. Recent real-world analysis of WebRTC deployment shows that direct peer connections and server-reflexive paths succeed in approximately 75-80% of consumer Internet sessions, with the remaining 20-25% requiring TURN relay fallback [8].

Session Traversal Utilities for NAT (STUN) (RFC 8489) [9] allows clients to discover public IPs as stateless services. TURN (RFC 8656) [10] offers a fallback relay mechanism in cases when direct connections fail. This creates latency and bandwidth costs, but ensures that connection is alive. TURN relay services can cost approximately £75-£150 per month for typical bandwidth usage, adding a considerable amount to a total operational cost for systems aiming to minimise central infrastructure dependency [11].

2.3 Transport protocols

Traditional remote access protocols work over Transmission Control Protocol (TCP) because it is more reliable by design [12], but introduces latency due to its delivery guarantee. Studies show that TCP's congestion control, defined in RFC 5681 [13], can add 300ms latency when a connection is experiencing packet loss.

Quick UDP Internet Connections (QUIC) (RFC 9000) [14] combines TCP's reliability, User Datagram Protocol (UDP) performance and Transport Layer Security (TLS) 1.3 [15] encryption at the transport layer. Research [14] indicates that QUIC reduces connection establishment time by 30-40% compared to TCP+TLS configurations, more prominently visible in high-latency or packet loss conditions, although benefits vary significantly depending on network conditions. The trade-off for QUIC is infrastructural challenges against UDP traffic and higher CPU overhead compared to TCP.

2.4 Cryptographic frameworks

TLS 1.3 (RFC 8446) [15] underpins the system's security with mandatory perfect forward secrecy. Perfect Forward Secrecy (PFS) is a cryptographic property that prevents long-term key compromise from affecting past session keys. This is achieved through ephemeral key exchange mechanisms such as ephemeral Diffie-Hellman (DHE) or elliptic curve Diffie-Hellman (ECDHE), where each session uses a unique, short-lived key pair that is discarded after use [16]. The Noise Protocol Framework [17] offers alternatives specifically for peer-to-peer scenarios, providing modular cryptographic patterns enabling mutual authentication without certificate infrastructure.

Argon2 (RFC 9106) [18] provides strong password hashing that is resistant to GPU attacks. It can be adapted for specific threat models, therefore will increase the offline attack costs compared to bcrypt [19] or PBKDF2 [20].

2.5 Privacy Design

Privacy-preserving architectures are based around design principles where central servers are prevented from accessing user data or communication. This can be accomplished by introducing ephemeral credential storage with short time-to-live (TTL) intervals and strong cryptographic commitments, potentially adopting quantum-resistant schemes for future resilience [21].

This becomes crucial when considering cryptographic design principles for high-risk environments, including state-backed attackers with retrospective surveillance capabilities, where forward secrecy and credential lifecycle management are essential security controls [16].

2.6 Platform Considerations

Cross-platform compatibility may be challenging due to each platform's different security models. Windows provides a rather easy screen capture via Desktop Duplication Application Programming Interface (API) [22] and input via SendInput [23]. Android's MediaProjection API [24] also provides good capture and control, but requires explicit user consent.

Conversely, iOS has the most restrictive environment. Screen capture is only allowed on Mobile Device Management (MDM) enrolled [25] devices or through private APIs that cannot be used in applications downloaed from app store App Store, complicating the publication [26]. This project addresses iOS constraints through view-only screen sharing.

3 Technical Stack Justification

3.1 Backend: Rust

The selection of Rust [27] as a base implementation technology is based around deep analysis of memory safety requirements and performance features. Memory safety vulnerabilities contribute to over 70% of security issues in systems software [28]. Rust's ownership mechanism enforces memory safety at compile time without a garbage collection overhead. Recent wide adoption of Rust demonstrates practical impact across industry [29, 30]. Google's comprehensive shift to memory-safe languages, including Rust, contributed to a significant reduction in Android's memory-related vulnerabilities from 76% (2019) to 24% (2024) [31].

Rust's concurrency model prevents data races at compile time, making parallel processing safe and efficient. Asynchronous syntax and mature runtimes like Tokio [32] give convenient concurrent programming that matches C++ speed without race conditions and deadlocks. Rust's ecosystem provides a vast library of crates (libraries) for cryptography (rustls, ring) [27], networking (quinn, tokio) [33], serialisation (serde) [34] and the whole WebRTC stack (webrtc).

Alternative languages were also considered but ultimately rejected. C++ offers great performance, has one of the best framework libraries available on the market and is widely adopted in the industry, but requires constant vigilance over memory safety [30]. Go [35] lacks a well-established desktop capture API, there are only some open-source projects that are not used in commercial software. Node.js is single threaded, so it would be impractical to use in software that required complex CPU operations like encoding and encryption/decryption.

3.2 Frontend: Flutter

Flutter [36] enables sharing of substantial source code across Windows, Android and iOS with platform-specific User Interface (UI) conventions, exactly what the project is aiming for. Dart [37], the language that powers Flutter, compiles directly to native machine code using its own engine, reducing memory usage. Unlike JavaScript, Flutter has much lower overhead with frame times typically within 10-15% of platform-specific frameworks. A plugin architecture makes it easily integrable with platforms' native APIs and allows simple cross-platform development.

Qt [38] with C++ offers great multimedia support but presents difficulties with licensing for commercial use, offering limited functionality under open-source license [39]. Users report that it is rather simple and useful, but this is mitigated by the C++ memory management complexity. React Native [40] has support for vast number of libraries, although there are complications with adapters, but it eventually was eliminated due to large performance overhead and limited desktop support. Electron [41] was rejected for massive binary sizes that are disproportionally big compared to what they offer, along with significant memory consumption overhead compared to native frameworks.

3.3 Interoperability: Flutter Rust Bridge

Flutter Rust Bridge (FRB) [42] is automatically generated using cargo (build system and package manager). It provides automated two-way communication between Dart UI and Rust logic. This removes the need for manual Foreign Function Interface (FFI) [43] maintenance and eliminates integration complexity. FRB generates type-safe Dart interfaces for Rust APIs, that will automatically handle memory management.

Performance results match the requirements for real-time applications. The Flutter side can receive streams directly from Rust async iterators running in the background, enabling efficient data transfer without unnecessary memory copying.

3.4 Supporting Infrastructure

Redis [44] serves as a session store for its performance. It runs in Random Access Memory (RAM), that makes it faster than any persistent data storage. It also offers built-in Time-To-Live (TTL) support that aligns with the ephemeral credential design that was chosen. The Key-Value model maps to session management where room IDs and tokens serve as keys.

Axum [45] serves as the Hypertext Transfer Protocol (HTTP) framework, selected for its strong performance and type-safe routing. It also has good integration with Tokio, allowing for smoother, less complex development. Benchmarks indicate strong performance characteristics suitable for handling concurrent connections.

Tokio provides the asynchronous runtime underpinning of all input/output (I/O) operations. It distributes work across all available CPU cores by allowing concurrent connection handling with minimal context switch overhead.

4 System Architecture

4.1 Architectural pattern

The current implementation is based around a three-tier architecture separating client applications, signalling and state storage [46]. The design allows solid scalability without the overhead of a Microservices architecture. It separates the concerns and accelerates development speed by allowing independent development of tiers. The stateless signalling server can scale horizontally via load balancing, while session state remains accessible from multiple Redis server instances.

4.2 Session Establishment

The Room-based model implies one peer to be an Initiator and the other is a Receiver. The initiator creates a room via the signalling API, that returns a room Identifier (ID) and password. For now, these need to be communicated to the Receiver through a third-party channel. Future implementations will explore QR code scanning, deep linking, and cryptographic protocols for key exchange to eliminate this out-of-bounds credential exchange requirement.

When the room is created, the APIs generate room credentials and store them in Redis with a 5-minute TTL. The Initiator receives a unique session token that is used for status checks and room management. When the responder provides valid credentials, the room state is changed to connected and the responder receives a session token.

4.3 Data Flow

The interaction between the Flutter clients and the backend infrastructure follows a specific sequence of HTTP-based signaling actions. As illustrated in Figure 2, these actions are numbered to correspond with this lifecycle:

1. **Registration (Action 1):** Both the Initiator and Responder post a device label to the Rust signaling server. After this, they receive a client Universally Unique Identifier (UUID), session token, display name and heartbeat intervals.

2. **Heartbeat (Maintenance):** The client then periodically updates its status to allow the server to renew timestamps and manage sessions.

3. **Room Creation (Action 2):** The Initiator posts credentials to the server. The server issues a room ID, password, initiator token and

Time-to-Live (TTL), then stores these in Redis with AES-256-GCM encryption.

4. **Status Polling (Action 4):** As shown by the orange dashed line in the diagram, the Initiator continuously polls the server until the Receiver successfully joins.

5. **Room Joins (Action 3):** The Receiver enters the room ID and password. The server verifies the hash, emits tokens, sets the joined flag and deletes the room metadata to finalise the handshake.

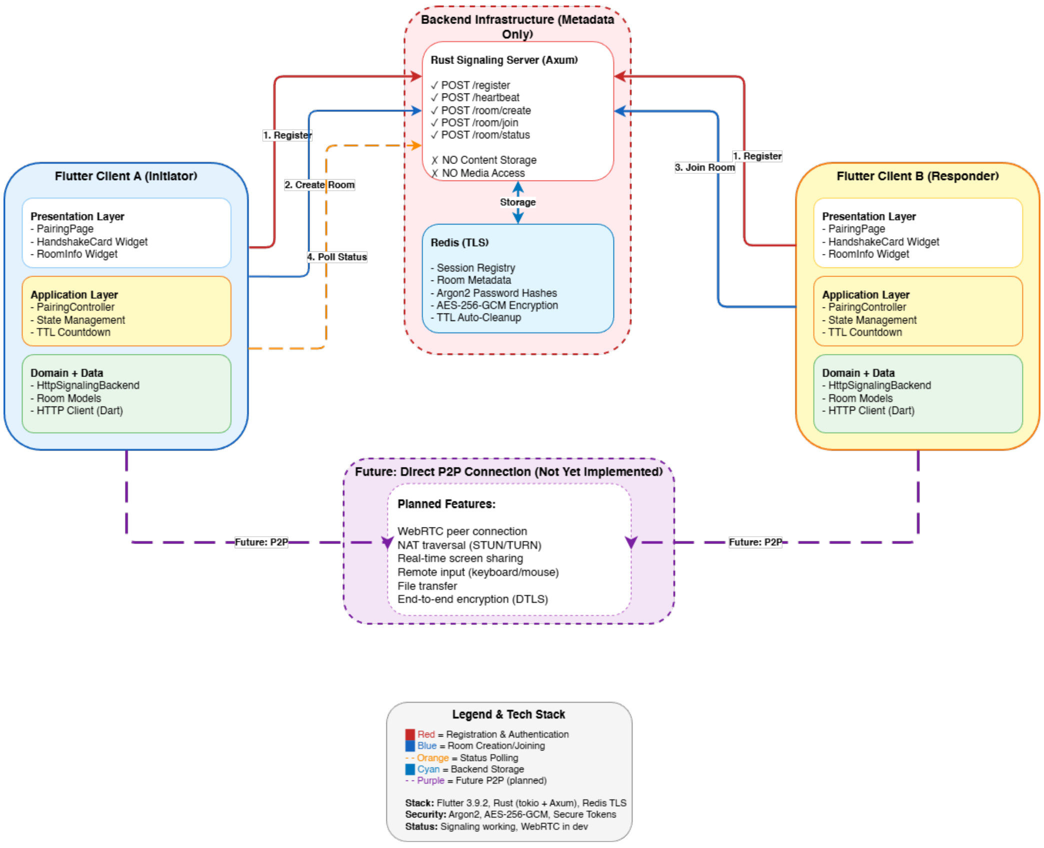

Figure 2: Diagram of the current state of the System Design and future plans.

The system architecture has developed from the initial high-level idea, moving to a current working prototype. Figure 1 demonstrated the first attempt to model the system based around minimal server involvement, where the signaling server handled only metadata. Figure 2 reflects the system as it exists now and plans for near-future development, where peers would eventually communicate directly using WebRTC-based NAT traversal over different networks. The new version retains the same high-level structure but using HTTP-based signaling, Redis-backed session storage and polling-driven updates to coordinate connections. At this stage, the signaling server manages peer discovery and session status with plans for direct P2P transport for future iterations.

4.4 Current Limitations

The current implementation is restricted to same-network connectivity, which is the most significant limitation. The prototype successfully establishes connection from Client to Signalling Server when peers share a local network with direct IP routing. Later stages will upgrade the connection to handle cross-network connectivity. The other issue is out-of-bounds credential exchange, which adds extra friction to the process and complicates the connection.

Finally, polling-based status checking adds avoidable latency and server load, compared to WebSocket or Webhook real-time updates. Currently clients poll the status every second, which can introduce approximately 500ms delay between connection state change and client awareness, as polling mechanisms exhibit inherent latency due to the time interval between successive requests.

5 Roadmap and Next Steps

5.1 Cross-network connectivity

The most important milestone for the nearest future is developing cross-network connectivity with WebRTC-based NAT traversal. This will be separated into three stages for different candidate nomination. It will start with STUN integration for server-reflexive candidate discovery, then expand to ICE candidate exchange and finish with TURN relay as a fallback for signalling infrastructure. Initially the STUN will use public servers with future potential to migrate to a self-hosted server ready for production.

To satisfy the needs of ICE candidate exchange, the signalling logic will need to be refactored for bi-directional transmission between peers. The current polling mechanism will be reconsidered in favour of a more robust solution, that will enable users to search for new candidates until a connection is made. Implementation will use Rust's webrtc crate that provides support for all ICE, STUN and TURN.

5.2 Screen Capture and Transmission

After successful cross-network connectivity implementation, the development will shift towards real-time screen capture and transmission. The compression and transmission logic would be shared, but the capture logic would be platform specific with their appropriate requirements. Windows and Android do have relevant APIs to utilize for such tasks, although Android will require user consent for screen capture. iOS does not allow third-party screen capture, so it will have to rely on view-only functionality.

The capture frames require compression in order to be transmitted over the network with an acceptable bandwidth consumption and latency. Initial implementation will explore JPEG compression as it offers acceptable results with little complexity. Later the project may compare performance of codecs such as AV1 [47] and VP8/VP9 [48] to determine which offers better performance in benchmarking tests.

To transmit the data, the implementation will test a solution with UDP-based approach for minimal latency. It could later potentially migrate to QUIC for improved packet loss recovery and congestion control. It will also include adjusting frame rate and compression quality to adapt for network capabilities. Performance target for a standard broadband connection speed (10-11Mbps) is 30fps and end-to-end latency under 100ms.

5.3 Remote Input and Control

Remote input implementation will allow a Receiver to manipulate the Initiator's system via cursor and keyboard input over the peer connection. There would be different input injection specific requirements for the platforms, like SendInput on Windows and AccessibilityService on Android.

To reinforce the security, there will need to be toggleable clipboard and restricted system commands execution to prevent potential abuse. The system will require explicit user consent before allowing input from remote host, with appropriate UI indicators that display a remote control status. Users will have an ability to instantly deny permissions for remote input using an emergency hotkey or a UI gesture.

5.4 Evaluation and User Surveys

The evaluation stage will assess the prototype across multiple performance, security and usability indicators. Performance would be measured for NAT traversal success rates across different network conditions and setups, latency, bandwidth and power consumption. Security aspect will evaluate impenetrability and resistance from man-in-the-middle attacks, session hijacking, credential enumeration and remote code execution. The testing will also extend to secure key handling and data storage.

In the later stages of the development, there are plans to conduct a set of surveys and user evaluations to gather relevant feedback from representative target audience. The study will be conducted in accordance with Edinburgh Napier University academic and privacy guidelines with explicit written consent of participants. Participants will complete realistic scenarios to help determine usability issues and collect measurable feedback. The results will assist with final improvements before the project submission to provide a higher quality prototype and build a roadmap for further development.

6 Conclusion

This interim report shows the progress of initial research and prototype development of a secure peer-to-peer remote access system, key focus of which is demonstrating viability of cloud-minimal approach. The core design choices and infrastructure development in the second stage of the project established a foundation for completing the remaining part of the work. Selected technical stack, Rust and Flutter, provides a memory-safe, cross-platform solution necessary for production-level code quality. The remaining part of the project will focus on completing cross-network connectivity, screen capture and transmission, as well as remote input control. The prototype will provide a system, that demonstrates peer-to-peer remote access concepts and will prove that alternative P2P approach is competitive with cloud-centric solutions.

The project addresses the needs of privacy-conscious users, many of whom may have a higher risk of being targeted. This includes journalists and NGOs in hostile environments, where commercial surveillance is a major risk. By showing that P2P architecture can provide practical remote access without heavily relying on the central server, this work contributes to a more global goal of developing privacy-preserving alternatives to centralised internet services.

References

[1] Microsoft, “Understanding remote desktop protocol (rdp) - windows server,” 2023. Microsoft Learn.

[2] T. Richardson and J. Levine, “Rfc 6143: The remote framebuffer protocol,” 2011. IETF Datatracker.

[3] K. S, “100ms.live blog,” 2022. 100ms.live.

[4] A. Begen, M. Saleem, and S. Shaikh, “Rfc 8866: Session description protocol (sdp),” 2021. IETF Datatracker.

[5] V. Fayed and V. Giotsas, “One ip address, many users: Detecting cgnat to reduce collateral effects,” 2025. The Cloudflare Blog.

[6] S. Sharwood, “Isps more likely to throttle netizens who connect through carrier-grade nat: Cloudflare,” 2025. The Register.

[7] A. Keränen, C. Holmberg, and J. Rosenberg, “Rfc 8445: Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal,” 2018. IETF Datatracker.

[8] Getstream.io, “Learn stun turn servers on webrtc,” 2025.

[9] M. Petit-Huguenin, G. Salgueiro, J. Rosenberg, D. Wing, R. Mahy, and P. Matthews, “Rfc 8489: Session traversal utilities for nat (stun),” 2020. IETF Datatracker.

[10] T. Reddy, A. Johnston, P. Matthews, and J. Rosenberg, “Rfc 8656: Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun),” 2020. IETF Datatracker.

[11] A. Alakkadshaw, “Turn server costs: A complete guide,” 2023. DEV Community.

[12] I. S. Institute, “Rfc 793: Transmission control protocol,” 1981. IETF Datatracker.

[13] M. Allman, V. Paxson, and E. Blanton, “Rfc 5681: Tcp congestion control,” 2009. IETF Datatracker.

[14] J. Iyengar and M. Thomson, “Rfc 9000: Quic: A udp-based multiplexed and secure transport,” 2021. IETF Datatracker.

[15] E. Rescorla, “Rfc 8446: The transport layer security (tls) protocol version 1.3,” 2018. IETF Datatracker.

[16] NIST, “Guidelines for tls implementations (sp 800-52 rev. 2),” 2023. National Institute of Standards and Technology.

[17] T. Perrin, “The noise protocol framework,” 2018. Noiseprotocol.org.

[18] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, “Rfc 9106: Argon2 memory-hard function for password hashing and proof-of-work applications,” 2021. IETF Datatracker.

[19] B. Buchanan, “bcrypt or argon 2?,” 2024. Medium.

[20] S. Team, “Argon2 vs bcrypt vs. scrypt: which hashing algorithm is right for you?,” 2023. Stytch.

[21] NIST, “Post-quantum cryptography standardization,” 2024. National Institute of Standards and Technology.

[22] Microsoft, “Desktop duplication api,” 2022. Microsoft Learn.

[23] Microsoft, “Sendinput function (winuser.h),” 2022. Microsoft Learn.

[24] A. Developers, “Media projection,” 2025.

[25] A. Support, “Device management restrictions for iphone and ipad devices,” 2025.

[26] I. S. Team, “New block screen capture for ios/ipados mam protected apps,” 2025. Microsoft Tech Community.

[27] B. Buchanan, “Rust and cryptography,” 2021. Asecuritysite.com.

[28] C. Cimpanu, “Microsoft: 70 percent of all security bugs are memory safety issues,” 2019. ZDNet.

[29] I. Novytskyi, “Rust adoption guide following the example of tech giants,” 2025. Xenoss.

[30] Microsoft, “We need a safer systems programming language,” 2019. Microsoft Security Response Center.

[31] Google, “Eliminating memory safety vulnerabilities at the source,” 2024. Google Online Security Blog.

[32] T. Team, "Tokio - an asynchronous rust runtime," 2025.

[33] Leapcell, "When to use async runtimes in rust concurrency and when not," 2025. Leapcell.io.

[34] M. Endler, "The state of async rust: Runtimes," 2024. Corrode Rust Consulting.

[35] GoLang, "The go programming language," 2025.

[36] F. Team, "Multi-platform," 2024. flutter.dev.

[37] ThinkPalm, "Why flutter is the most popular cross-platform app development sdk?," 2024.

[38] Q. Group, "Qt licensing - choose the right license for your development needs," 2024.

[39] Q. Group, "Qt - obligations of the gpl and lgpl," 2025.

[40] B. Ranaweera, "React native vs flutter: Which saves more development time in 2025?," 2025. Blott.com.

[41] T. Federico, "Why electron is a necessary evil," 2021.

[42] C. Yang, "Flutter rust bridge: Flutter/dart ¡-¿ rust binding generator," 2025. GitHub.

[43] R. Documentation, "Ffi - the rustonomicon."

[44] Redis, "Redis persistence," 2025.

[45] Leapcell, "Rust web frameworks compared: Actix vs axum vs rocket," 2025. DEV Community.

[46] M. Chiaramonte, "Understanding the architecture of a 3-tier application," 2024. vFunction.

[47] rust av, "Av1an: Cross-platform command-line av1 / vp9 / hevc / h264 encoding framework," 2025. GitHub.

[48] T. Ruether, "Video codecs: What they are & the best formats for streaming," 2024. Mux.com.