University of Glasgow | School of Computing Science

Onion-Routed Communication Over WebRTC

Kyle A. Simpson, 2029567s

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 28, 2016

Abstract

In light of various reports and leaks concerning the monitoring powers of nation-states and other well-funded adversaries, privacy on the internet has become a hot topic in recent years. However, state-of-the-art defences such as onion routing have not seen adoption in general usage outside of enthusiast circles and in cases of dire urgency. Designing a small-scale system to use this technology from the web browser environment may potentially address this lack of uptake by increasing usability. Additionally, the ability to use onion routing selectively may be useful for application developers who desire higher performance for certain classes of traffic.

I provide the design of a simple privacy-focused messaging application, as well as the design and implementation of a suitable network stack providing features necessary for secure applications and communication over WebRTC. I believe that this shows the viability of introducing small-scale onion routed paradigms into conventional networked applications to provide increased security, by treating the application server as a nominated peer.

Acknowledgements

I would like to thank Dr. Colin Perkins for helping me to develop my project proposal, as well as for his continuous support, guidance and advice throughout the year.

Additionally, I'd like to thank my family, my girlfriend Vanessa, and my friends for listening to me prattle on about network topologies more than anyone should have to! In all seriousness, their support has been invaluable this year. I'd like to thank my father specifically for taking the time to read through and proof read my report, despite the length, subject matter and inconvenience.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

In recent years, privacy online has been a growing concern for many users who fear interception of their internet traffic by sufficiently determined and well-funded threats such as criminals or nation-states. Packet analysis of typical internet traffic can reveal information about which services a user accesses over the internet, who they communicate with, as well as the contents of the data sent across the internet if encryption is not used. Onion routing is widely believed to be the best defence against such adversaries, and is often used to hide activities both benign and criminal.

However, usage of Tor and other such clients is often dramatically slower than standard internet traffic, and it is seen as extremely unwieldy to set up for specific programs. As a result, very few users make use of the technology to secure their communications. A specifically tailored, cross-platform application could utilise modern web technology and onion routing techniques to make it drastically easier for users to stay in touch without fear of their privacy being breached.

One such environment that provides the safe, cross-platform execution we desire for making onion routing easily deployable is the web browser. However, peer-to-peer connection schemes such as WebRTC remain in an experimental state. The core problem is to assess whether these peer-to-peer technologies are mature and dependable enough to build an onion-routing protocol into a web-browser. Perhaps more importantly, this should address the deployability issues of Tor and related systems by occurring entirely within the browser sandbox and without requiring the user to install extra plugins or extensions – by requiring no user effort.

In this report, I show the feasibility of providing onion-routed communication within the standard browser environment by presenting an implementation of an onion-routing scheme over a heavily modified variant of the Chord peer-to-peer network. However, the inconsistent state of WebRTC support across different browsers heavily limits the effectiveness of my design. Additionally, I show how modifying the Chord protocol to be message-driven instead of exclusively file-oriented enables the design of modern, extensible systems, and that the addition of a cryptography-based authentication and identification schema with strong file ownership allows for the design of secure applications. Furthermore, the paper presents an onion-routing algorithm designed for environments where Transport Layer Security is not practical or usable, and shows the required design considerations for implementation of Chord in an environment with strong restrictions on connectivity.

The remainder of the report is structured as follows: **Chapter 2, Background** will provide a brief explanation of various concepts which underpin the report's content, such as onion routing, WebRTC and the Chord peer-to-peer protocol. Additionally, it supplies a light discussion of related work which I have encountered during my research for this project. **Chapter 3, Requirements** contains a high level description of the functionality required by the application and network stack, as well as the assumed capabilities of an adversary that the system is designed to defend against. **Chapter 4, Architecture and Design** describes the structure and high-level operation of the complete system, from the application design to the mechanisms of the network stack. Also discussed

is the evolution of my system design as new requirements have arisen. **Chapter 5, Implementation Details** discusses the network stack implementation from a lower-level perspective, introducing the various difficulties and roadblocks encountered during development as well as the methodology and limitations of any operational testing. **Chapter 6, Security Considerations** provides a discussion of how each layer affects the security of the overall end project, along with various attacks proposed on the presented design. **Chapter 7, Evaluation** considers my thoughts on the design presented in this report after its implementation, as well as my experiences throughout the year working with WebRTC, deploying to multiple platforms and working with JavaScript. **Chapter 8, Discussion and Future Work** concludes the report, summing up and reiterating my findings from this project as well as discussing improvements to the designs and material discussed here to make the system examined more suitable for production deployment.

Chapter 2

Background

This section introduces several topics which are important in the context of this report: WebRTC, Chord and Onion Routing.

Web Real-Time Communication (WebRTC) brings true peer-to-peer connection between browsers, and is thus the main form of connectivity used throughout the project. Its connection model differs significantly from more common models such as TCP or any other sockets based approach – as such, it is worth explaining the operation and design rationale of the protocol. *Chord* is the basis for the network topology used in achieving decentralised onion-routing; it provides key-based routing, a high degree of reliability and is known to scale well. Construction of its hallmark ring topology is described, as are some alternate variants of the protocol. Finally, *Onion Routing* is the key topic of this report, and is a cryptographic procedure to protect data from external agents by applying successive layers of encryption. It is introduced and explained with reference to two well-known implementations of the concept - *Tor* and *I2P*.

Additionally, a selection of papers related to these topics and to the report in general are introduced and discussed briefly. Their contributions to the ideas in this report are mentioned, as are any significant differences.

2.1 Web Real-Time Communication (WebRTC)

WebRTC is a protocol standard and browser API which allows any two compatible web browsers to communicate directly with one another. Browsers discover partners and exchange connection data with the aid of a *signalling server*, as seen in **Figure 2.1**. Additionally, it takes care of other obstacles for peer-to-peer communication such as Network Address Translation (NAT), a form of IP indirection, by trying multiple routes to the selected partner and choosing the best available path.

WebRTC is a multimedia-focused peer-to-peer protocol designed to support and enable Voice over Internet Protocol (VoIP), video conferencing, efficient peer-to-peer online gaming and file transfer [9]. Proposed in response to the vast quantity of proprietary, plugin-based implementations of direct communications systems in web browsers, WebRTC was born in part from the lofty goal of standardising direct multimedia transfer between interoperable browsers and other capable endpoints. The protocol focuses primarily on audio and video communication between two compatible user agents, with binary data transfer remaining a secondary (but important) concern. Remarkably, much of the protocol's design is driven by a desire to make use of existing, established protocols to aid in the adoption process and to provide a degree of backward compatibility where possible [5].

The WebRTC protocol provides two classes of peer-to-peer connection – Media Streams and Data Channels, for audio/video streaming and raw data respectively [8]. However, if one or more of the users are behind firewalls

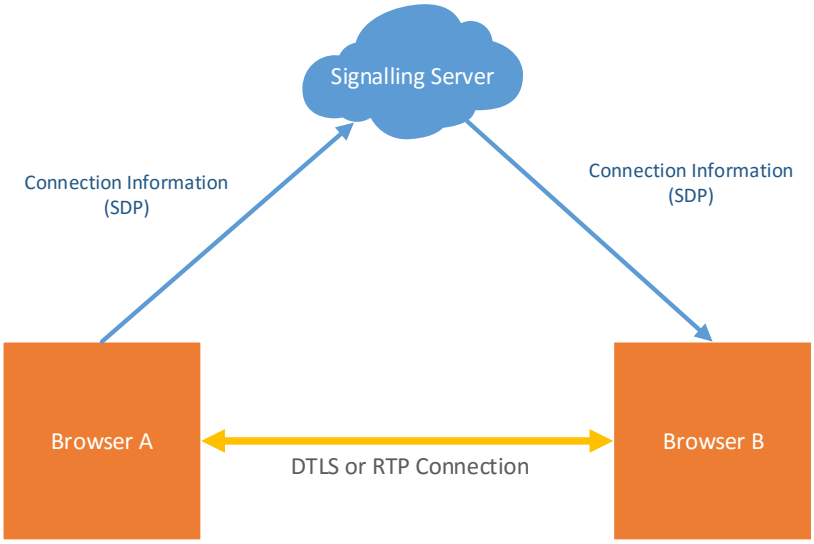


Figure 2.1: *High level operation of WebRTC.*

or Network Address Translation (NAT) then this can prevent or seriously impede communication. To provide peer-to-peer functionality within and between browsers, WebRTC makes use of the User Datagram Protocol with the help of Interactive Connectivity Exchange (UDP+ICE) to provide NAT traversal for all peer connections [10].

Security lies at the heart of WebRTC's design. In media streams, all data is carried using the Secure Real-Time Protocol (SRTP), and across data channels Datagram Transport Level Security (DTLS) is used. These choices are well-founded – DTLS is a derivative of the Secure Sockets Layer protocol (SSL) (which is known to be secure), while RTP has been proven more than capable for media transfer since its inception in 1996. SRTP represents the addition of cryptographic properties and secure profiles to the RTP standard to achieve authentication and data protection. For both media streams and data channels, an initial key exchange is performed through *DTLS-SRTP keying*. Specifically, these technologies add confidentiality, integrity protection and source authentication to all of WebRTC's transports [10][17]. This makes WebRTC extremely suited for use in secure contexts as well the standard use cases.

Data channels then implement the Stream Control Transmission Protocol (SCTP) on top of DTLS to provide a high degree of configurability for various transport guarantees. For instance, data transfer may be unreliable (like UDP) or reliable (as in TCP) as needed by the application layer, and data channels may or may not have in-order delivery guarantees. This configurability is extremely important for implementing different classes of application with WebRTC – consider most video games, which do not require guaranteed in-order delivery, in comparison with file transfer or IRC-like text messaging clients. Additionally, the particular order of these protocols in the layer stack was chosen to allow transmission of arbitrary-size binary data over data channels [10].

2.1.1 Connection Model

Establishment of a WebRTC Peer Connection relies on out-of-band exchange of Session Description Protocol (SDP) and ICE data. This model is heavily influenced by textual protocols such as Session Initiation Protocol (SIP), which is designed to convey information about a host's capabilities such as their supported codecs or preferred transport guarantees. In fact, WebRTC allows for the use of *any* potential signal channel [1]. The

inclusion of this model is particularly important for negotiating streaming and content capabilities for Media Streams, as RTP is built with the assumption that an external signalling channel like SIP is in use [17].

WebRTC makes use of SIP-like offer/answer semantics, as browsers are otherwise unable to establish direct connections to one another [1]. When opening a connection, a client generates an *SDP offer*, containing a list of their capabilities, DTLS parameters to help establish a secure connection, and potentially several ICE candidates. This is sent along a signalling channel to an external server, who is responsible for directing the offer to another client. On receipt of an SDP offer, another client generates an *SDP answer*, taking into account their partner's capabilities as well as their own before sending the reply across the signalling channel. During this exchange additional ICE candidates may be transferred through a process known as *ICE trickling*. Once the initial client receives the SDP answer and enough ICE candidate data has been exchanged, the clients determine the best available route and the connection is opened.

Critically, WebRTC imposes no constraints on encryption of signalling channels – SDPs and ICE leak the true IP address of each node to the outside world if developers are not careful, as they are under no obligation to encrypt or otherwise make their signalling channels secure. This in stark contrast to the design of the connections themselves, which encrypt each and every packet to protect user communications. As a result of this, the greatest vulnerability to user security arises from careless developers who may either be unaware of this fact or make a mistake during the implementation of their signal channel [1].

Due to this, when signalling to a potential connection partner it is worthwhile to obtain some "fingerprint" connecting a partner's key-pair to their identity in addition to encryption of SDP and ICE data – this functionality is typically supplied by an Identity Provider (IdP). This guarantee is by no means necessary, but WebRTC can interact with IdPs and include verification data alongside SDP or ICE candidates to automatically protect against man-in-the-middle attacks by utilising the externally trusted service for verification and authentication [19]. This project does not make any use of these features, instead directly tying the concepts of verification and authentication together by making use of *self-certifying identifiers*, discussed further in section 4.3.1.

2.1.2 Network Address Translation (NAT) Traversal

One of the key services that WebRTC provides is NAT Traversal. Many users connect to the internet behind one or more levels of IP indirection: from inside of a home router, within their Internet Service Provider to divide IP addresses amongst multiple clients or elsewhere between a user and their destination. This can make it very difficult for a machine on the outside of a NAT to directly contact a user on the other side – tunnels through NAT must be opened by the internal user, and are often ephemeral in nature. WebRTC makes use of Interactive Connectivity Exchange (ICE) to allow any two clients to gather a list of routes which they believe themselves to be known by, before exchanging these routes with their partner through an external server. Each client then tries each supplied route in turn, determining which are valid and which are not, and both clients arrive at a consensus, choosing the best route for communication across NAT. ICE address gathering is helped by two companion protocols: Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN).

STUN is a client/server model protocol used to help two clients communicate across NAT. Although it is a key tool for NAT traversal, it is *not* a complete solution by itself, and is used as part of ICE and other complete schemes. STUN's main use is for a client to determine whether it is behind one or more layers of NAT, and to obtain an externally usable IP address in the event that this is the case. Additionally, STUN can check connectivity between two end points, and can be used as a keep-alive protocol to maintain NAT bindings. To do this, a STUN client contacts a STUN server to determine which "binding" a NAT has allocated to it. As this packet travels across the internet, it may pass through multiple NATs, modifying the source address and port each time. When the STUN server receives this request, it takes note of the source IP address and port of the request packet – which will correspond to the binding on the last NAT that the packet travelled through. This is known as the

server reflexive address, and is effectively the client's IP address as seen by the outside world. The STUN server replies to the client with this information in an obfuscated manner. The client, on receiving this reply, now has knowledge of an external IP address that may be used to contact it – it may also determine whether or not it is behind at least one NAT if this address does not match its outgoing route [16].

The purpose of TURN is to help two clients participate in peer-to-peer communication in the event that one or both of the hosts are behind symmetric NAT, or when a direct communication path cannot be found for some other reason. To do this, a TURN server is used, acting as a relay for all packets sent to and from one of the hosts. The host behind the relay is capable of sending packets to control the relay, as well as to exchange packets with their peers by exposing one or more public addresses via the server. This operates through the client contacting a TURN server. The client then obtains an IP address and port on the server - known as a *relayed transport address*. The client then informs any other hosts that *this* is its address, allowing communication to occur. The server then overwrites the source field of the headers of any traffic from the source to originate from this new address, and similarly replaces the relayed transport address of any incoming packets to redirect them to the true destination.

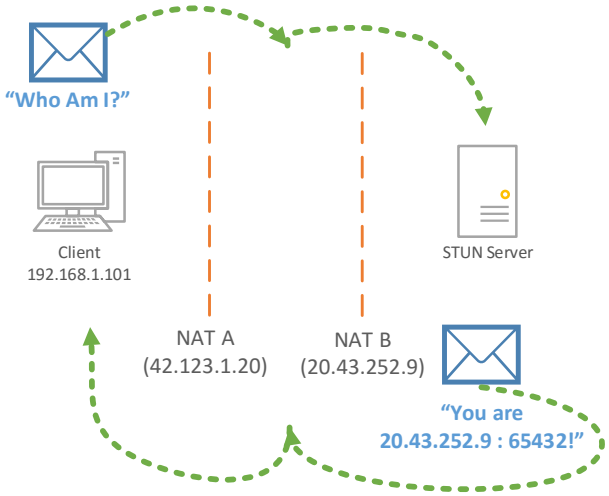


Figure 2.2: *Simplified model of how STUN works to provide the server-reflexive address.*

Naturally, this comes at high cost of bandwidth to the TURN server, especially where many users must be served or in the case of video streaming. TURN servers are typically rented or paid for to provide this privilege as a consequence of the additional load they impose on the resources of a machine and its network [15].

In the context of usage within WebRTC, relaying packets in a connection has no impact on the security implications considered above. This is due to the use of SRTP or DTLS to encrypt media streams and data channels respectively – the TURN server is only able to parse the UDP headers of any packet in the flow. Although useful in practice, TURN is not utilised as part of the project. This is largely due to the additional load (and cost) this would impose if I were to host such a server myself.

ICE is a protocol designed to provide NAT traversal for UDP-based sessions which utilise a SIP-like offer/answer model. Historically, offer/answer schemes have been hard to use behind NATs as they typically contain physical IPs of media sources and sinks – critically, NAT often makes it impossible to contact such addresses. As well as providing connectivity, we also desire to create a direct communication where possible to reduce latency, decrease packet loss and to reduce operational cost posed by application-layer intermediaries. ICE provides this functionality by coordinating STUN and TURN – obtaining a list of *ICE Candidates* from each host, a list of addresses which they know themselves to be reached by.

Clients which wish to open connections are known as *ICE Agents*. Each has a variety of transport addresses: physical IP addresses of their network interfaces, *server reflexive addresses* from STUN and *relayed transport addresses* from TURN. These are known as *Host Candidates*, and are categorised by the means they were obtained by. Each agent sends all of its available host candidates – either as a single transmission or piece by piece as they are gathered (*ICE Trickling*). The receiving ICE Agent then tries all candidates in parallel and sorts them into a priority listing based on: a) whether connectivity could be established, and b) the category of the candidate. Agents are expected to prioritise Physical over STUN over TURN in general. Once exchange is complete, each

ICE agent uses their generated priority lists to agree on the best suited candidates for communication, and the connection is opened once consensus is reached [20].

2.2 The Chord Protocol

In this project, the Chord protocol forms the basis of the overlay network for onion routing - it provides the reliability, efficient routing and global data lookup that are extremely desirable qualities in a distributed system.

Chord is a distributed lookup protocol developed by Stoica et al. to provide efficient data storage and lookup through key-based routing over a network of nodes [22]. One of the key points of Chord's design is that it enables message delivery across a massively scalable, geography-independent space while minimizing the amount of other machines that any node must know about. The algorithms used to maintain the ring are not documented here – these are the "stable" variants of the maintenance algorithms from the paper itself [22, p. 155].

To achieve this, all n nodes comprising the network are given a random m-bit key. Ideally, this achieves an even distribution across the space of 2^m identities for a suitably large choice of n - this is analogous to a mapping onto the the set of integers modulo 2^m. Each node must know of at least two other nodes to maintain consistency across the network topology: the first node preceding it in the identity space and the first node succeeding it. Respectively, these are known as the **predecessor** and **successor** of that node.

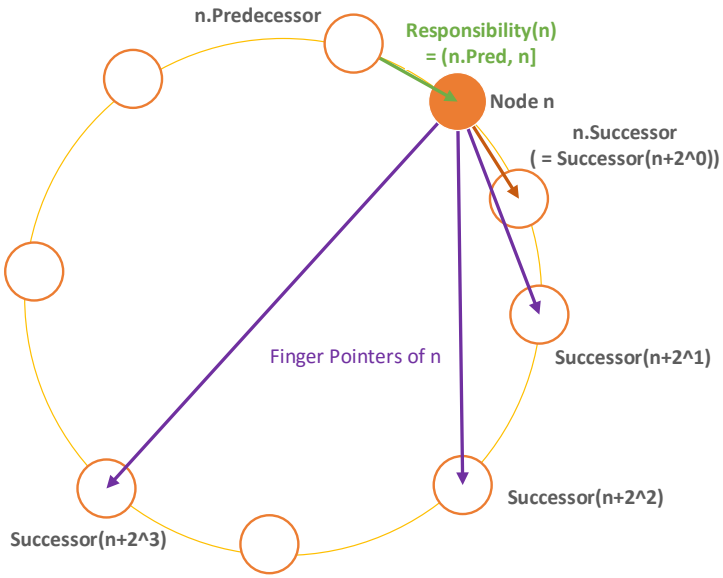


Figure 2.3: *Chord - showing file responsibilities and finger pointers.*

Any item may be stored in the network by using a suitable *consistent hashing* algorithm - i.e. one which minimises the amount of key-value pairs which must be remapped as the table's size grows and provides a relatively high degree of load balancing across the nodes. By taking the transform on any item, $\{Key, Value\}$,

$$Store(\{Key, Value\}) \longrightarrow \{h(Key), Value\}$$

for a suitable hash function h, we are able to determine which node within the system will store the item. Each node N within the system is assigned **responsibility over items located in the region** $(predecessor(N), N]$ **of the ring**; similarly to the above definition, N is said to be the successor of all keys in this region. This is why we ideally desire an even distribution of nodes across the identity space - by reducing the fraction of this space that a node is responsible for, it is less likely to have a higher load incurred from file storage, work distribution etc.

When choosing a hashing algorithm, Stoica et al. note that although cryptographic functions do not meet all of the required properties of a consistent hash function according to the original paper [12], their known intractability and collision resistance allow us to make reasonable guarantees of load balance given that an attacker cannot derive multiple keys so that they will overwhelm a specific node [22, p. 152].

It follows from the network topology that a node is guaranteed to be able to locate any stored item with a known key simply by following the successor pointers until the expected destination is reached. Suffice to

say, as the network scales to substantial amounts of nodes this incurs an unacceptably high amount of lookups throughout the system. To handle this, each node maintains a set of $log(m)$ look-aheads across the ring, known as the **finger table**. Denoting N as both a node and its assigned ID, we define for $i \in [0, m], finger_i = successor(N + 2^i)$ for N. It can be seen that successive finger pointers are further and further apart each time, "cutting across" the network and greatly reducing the amount of hops to locate an item. It is with this key optimisation that the network achieves $O(\log n)$ lookups with high probability for an n-node system - with the guarantee that an outdated finger table can still eventually lead to the destination due to the accurate successor pointers. While this optimisation increases the amount of other clients that a node must know of, it remains a minuscule fraction of the maximum identity space, while leading to generally massive performance gains.

Several variations of ring structured topologies similar to Chord have been proposed, most notably Koorde. As with Chord, Koorde nodes maintain active connections to their successor and predecessor - however instead of maintaining a finger table the look-aheads are based on De Bruijn graphs. Lookups in this scheme are based on bitshifts on keys to reach a destination, which produces $O(\log m)$ lookups for an m-bit key space. Additionally, more connections relative to total network size may be maintained to achieve a lower bound on maximum lookups [11]. Koorde, while theoretically faster overall is not as well-understood as Chord and so there is less literature regarding the stability of its different variants. In particular, Liu et al. found the standard variant of Koorde to hit "crash point" in a high-churn system considerably sooner than the standard Chord implementation [14]. It is unclear if the more stable variants of Koorde exhibit similar behaviour to Chord in these benchmarks.

2.3 Onion Routing

Onion routing is a cryptographic procedure to protect data from external agents as well as to provide anonymity to two correspondents in a communication flow. The technique works by successively encrypting data with multiple symmetric keys and sending it over a randomised path over relay nodes. Typically, each node is responsible for removing a layer of encryption and forwarding the packet to another host – with each node only knowing the previous and next hops within the route. This ensures that the true source of the data remains hidden from external analysis of the packets themselves, with only the start and end points of a route knowing any more information about the path.

Onion Routing has become somewhat famous in recent years due to the anonymity guarantees it provides, as well as among enthusiasts with a strong desire for privacy from their governments. Motivations for its use cover the full spectrum from personal reasons to the extremely political – in the case of whistleblowers, journalists, and those who fear retribution from their governments.

Although it is by no means computationally feasible to break the encryption used to protect the contents of onion routed packets, attacks exist nonetheless. Timing analysis is a known attack on Tor-like systems to determine whether or not communication is occurring between an external computer and some destination. If an adversary monitors known exit nodes in addition to a target suspected of using onion routing, then an attacker may notice consistent a consistent time difference between the transmission times of identically sized packets from the target and from an exit-node. This becomes even more detectable if an attacker can create congestion for the target, and see a corresponding change in delivery times from an exit node.

2.3.1 Tor Protocol

Tor is arguably the most well-known and most popular implementation of onion routing in use today – for means both illicit and honest. Tor's architecture is built around dedicated, central directory servers which contain a list of all available relay nodes. A node which wishes to establish a link obtains a list of available relays from a directory server, and chooses some random subset to act as the path to the desired destination. Obtained routes

guarantee a user perfect forward secrecy for all data sent across the channel. Importantly, Tor is a stream-focused protocol much like TCP.

Routes are built in Tor in a piece-wise fashion – a client begins a route by sending a build packet to the first hop along the route, informing them of their symmetric key and circuit ID to be used within the route. Subsequent build packets are encrypted using all communicated symmetric keys and are treated as regular onion traffic – appearing simply as relay packets until all layers of encryption are removed, allowing each hop in the route to discover the next hop and the address of the previous hop. This process repeats until the route is completed, at which point the last node is informed by an onion packet that it is the final node. Onion routes in Tor are bi-directional – once a link is opened, the source sends the list of symmetric cryptographic keys down the route for the end point to use. If the exit node needs to send data along the same link to the originator, then it applies the keys in reverse order and sends the data backwards along the chain.

Nodes in Tor communicate with one another using Transport Layer Security (TLS), and all packets within the network have equal size to attempt to prevent traffic analysis [7]. In practice, Tor is vulnerable to timing analysis by watching known exit and entry nodes and looking for patterns in the timing of packet entry and exit from Tor routes to look for correlations.

2.3.2 Invisible Internet Project (I2P) Protocol

In contrast to Tor, I2P makes use of UDP so is datagram- rather than stream-focused, and eschews the need for central directory servers by using peer-to-peer topologies to allow for distributed routing. Node IDs within the network are derived entirely from cryptographic data, employing the scheme of *self-certifying identifiers* within the NetDB router system – which also protects users from having to reveal their IP address to their communication partner. Routers within I2P actively profile one another, learning each others' capabilities, bandwidth and reliability to aid in the choice of efficient routes [25]. Additionally, I2P tunnels are unidirectional – every node exposes two inbound tunnels to the network, and encrypted packets are routed to the destination in a packet-switched rather than circuit-switched manner. Running multiple tunnels to and from source and destination endpoints also leads to a higher degree of resilience against failures by reducing the likelihood that node failures and exits will impact both directions of communication [23].

I2P employs a novel technique known as *garlic routing* in an attempt to address Tor's potential weakness to timing analysis. By bundling multiple messages together at different points in the network, giving each its own delivery instructions, I2P can introduce perturbations in message flows and paths as well as creating variation of delivery times [24].

2.4 Related Work

This report touches on a wide range of topics – as such, it draws inspiration from and develops upon some of the ideas expressed in a varied selection of research papers.

Caesar et al. discuss a clean-slate redesign of the internet, **ROFL**, in [4] using Chord rings within Autonomous Systems (AS) – a key aspect of their design is the use of *self-certifying identifiers* derived from cryptographic public keys to strongly link the concepts of authentication and verification of hosts. Additionally, their design makes use of Chord's segment responsibility rules to direct Internet Protocol packets to their intended hosts. These concepts form the basis for several of the presented modifications and enhancements of Chord. Notably, their model allows hosts to occupy multiple IDs to allow for multicast messaging – this is not accounted or modelled for within this report.

Another clean-slate design for the internet is explored by Liu et al. in [13] with the intent of integrating Tor's features into the base fabric of the network. By combining onion routers throughout ASs with *Rendezvous Mailboxes*, Liu et al. present an alternative design for the internet which they claim provides anonymity, security and denial-of-service resistance. The combination of mailboxes with onion-routed traffic underpins the communication model of the proposed messaging client, where these mailboxes are adapted to use a publish/subscribe model to handle groups instead of the standard put/get semantics they propose.

Avramidis et al. present **Chord-PKI** [3], a full public-key infrastructure built using the standard Chord protocol. It is designed to provide overlay network security (defence from attacks on the Chord protocol itself) while providing a strong trust-based basis for application security. This scheme allows for nodes to manage certificate granting, movement and revocation using sophisticated techniques such as threshold cryptography, clever state replication and network segmentation to defend against adversarial nodes. This system is far more complex, offering considerably more control of security features within the network. Additionally, usage of a full PKI decouples a node's ID from its certificates and keys, whereas the scheme explored in this report is far simpler – leaving aliasing of nodes to be handled by applications through usage of the file system. Avramidis et al.'s notion of a node's *trustworthiness* as derived from their perceived adherence to protocol is discussed as a potential future threat mitigation technique for the network stack presented here.

Dearle, Kirby, and Norcross discuss the application of Chord as a basis for fault tolerant services in [6]. Their approach treats a Chord ring as the host for a single distributed service – using the file system analogously to shared memory within a standard computer system. By splitting service components across multiple host nodes in combination with replication of state within the file system, they achieve an extremely high degree of fault tolerance by designing these components to be aware of the underlying data replication and properties of the network. Additionally, they hypothesise that their approach extends to key-based routing schemes in general.

2.5 Summary

Through this overview of related topics and necessary background for the report, I've shown how WebRTC is well-suited to the task at hand by enabling secure peer-to-peer communication within the browser. WebRTC's design handles a lot of the traditionally difficult elements of peer-to-peer communication, such as NAT traversal, capability exchange and route negotiation – additionally, the datagram communication semantics make its usage far simpler than TCP or other stream-based protocols, which require manual handling of packet framing at the application level. The protocol's design *does* require adaptation of any potential network stack to include a server which is in some way aware of the underlying peer-to-peer topology, due to its unique connection model. As a particular consideration, care must be taken surrounding the security and obfuscation of SDP and ICE candidate connection data to make sure that no unintended host can read this sensitive data inappropriately.

Following this, the Chord protocol was introduced – a ring topology of nodes built on **predecessors** and **successors** in combination with lookaheads across a circular identity space. The topology is very well suited to the design of the intended client by providing logarithmic-time lookups across the entire network *at any scale*. Most importantly for the remainder of the report, the concept of a node's **region of responsibility** within a Chord ring was introduced – while this applies purely to file storage in the traditional Chord model, this definition will be adapted to form the basis of a message-based variant of Chord.

Onion-routing itself was introduced, with an explanation of the benefits it ensures for communication such as endpoint anonymity and perfect forward secrecy. Discussing the paradigm with reference to Tor and I2P, two well-known implementations, provides some rationale on the design choices made when taking elements from each to provide a hybrid model. Finally, I introduce notable elements from related papers with a strong influence upon my final design – *self-certifying identifiers* to defend against identity spoofing within the network, and *rendezvous mailboxes*, which act as a denial-of-service resistant means of anonymous message delivery.

Chapter 3

Requirements

In this section, I introduce and discuss the high level requirements of the hypothesised messaging client - building up from an initial threat model where I discuss the capabilities of an attacker that the system must defend against, followed by the feature set that the messaging service must provide to users. From these two sets of considerations, I then derive a set of services that the network stack must provide to the application layer to make such a design feasible.

3.1 Threat Model

The threat model assumed here is an attacker capable of traffic analysis, determining the origin and destination of a given message. In general, they have three goals: viewing communication content, determining the participants of a conversation, and performing denial of service if these cannot be met. They are not assumed to be capable of breaking sufficiently strong cryptography, but they are assumed to be capable of spoofing their identity within a peer-to-peer system either through modifying packet headers or at the application level. Additionally, they are capable of modifying packets which pass over any node or computer they control, substituting fields or acting as an intermediary. This attacker has the resources to generate many identities within the network, and has the computational power, time, and money to devote to finding collisions against hashing algorithms which are known to be broken.

These capabilities are assumed for multiple reasons. Modern cryptographic algorithms rely on problems which are thought to be infeasibly hard to solve, such as large prime integer factorisation, i.e. there exists no known efficient algorithm to solve them. These form the basis of modern cryptography – if an attacker *did* have some means of breaking them at any arbitrary strength or key size, then no protection could suffice. *Assuming a limit to the attacker's capability is essential.* Reading headers of packets is, however, trivial – and in a peer-to-peer network, modifying them would be similarly easy. An attacker's ability to generate many identities is drawn from real-world experience; this technique is the crux of *Sybil attacks*, covered in detail in **Chapter 6**. Such attacks have been used against systems like Tor [2], as such my solution must account for (or at least be aware of) an attacker's ability to use these techniques against users. Several hashing algorithms, previously thought secure, have had powerful attacks developed against them. Methods of finding full collisions have been demonstrated against SHA-1, at budgets of 75K$ to 120K$ – this is stated by the authors as falling well within the resources of governments and criminal syndicates [21].

3.2 Messaging Client Requirements

The requirements for the messaging client were developed with reference to existing chat clients such as IRC, using the MoSCoW model of requirements classification. Requirements are categorised in terms of "Must", "Should", "Could", and "Would" in decreasing order of importance according to how essential each requirement is for providing the minimum level of core functionality.

3.2.1 Basic Interactions

A user *must* be able to begin a chat with any other user, or equivalently join a group.

This allows users to choose *who* they converse with, and allows users to meet one another on shared interests or topics of discussion. While well-known messaging protocols such as IRC take a "group-first" approach, enabling private messaging between two users *within* a group, due to the secure or confidential nature of conversations one-to-one correspondence should be an explicit aspect of the design.

Before a chat opens, users *must* be able to accept, decline or ignore the invitation.

This concerns one-to-one communication in particular. This is intended to act as a spam prevention counter-measure, ensuring that users cannot be inappropriately contacted or messaged without their express permission. Ignoring a chat offer *should* place the invite in some location on the interface where a user can return to it later – ignoring a second time would remove the invite *without notifying the sender*. Accepting, or explicitly denying a chat offer *should* notify the sender – this distinction gives users a level of control commonly seen in many messaging clients today.

A user *must* be able to send a discrete message to another user or group.

This is the basic communication primitive within the system – sending of simple strings (e.g. "Hi! How are you doing today?"). In both cases, if a message is sent without authorisation then it will not be seen by the recipient user or group. Messages must be tagged with the current identifier or nickname of the sender, as a means of identification within groups.

3.2.2 User Information and Features

Users *must* be able to assign themselves a nickname, provided that the given nickname is not in use.

Although users may wish to be identified purely through a random unique identifier (where true anonymity is desired), it is not practical for users to have to identify one another in this way.

Users *must* be able to create a group using an identifying name and a password.

Primarily, groups *must be identifiable to users*, and the inclusion of a password helps to act as an access control mechanism on which members are given authorisation to read message contents and post their own messages. Use of a password *should be* optional – not all groups will want to restrict their membership. Multiple groups may feasibly take the same name, as long as their passwords differ – this allows groups to have public and private sections of a chat channel.

Users *must* see a list of all (online) users who they have contacted or are in contact with in their current session. The network should have no concept of "friends".

Nicknames are, in practice, simply binding a temporary and memorable name to a similarly temporary (but unmemorable) random identity. For this reason, a user who talks to "BaconMan123" at one time may (at another time) either be talking to the same user with a different ephemeral identity, or to a completely different person. The ephemeral nature of user identities within the privacy model means, consequently, that it makes no sense for users to have a concept of friends within the client.

Users *should* be able to make themselves visible globally, advertising that they are available for discussion with any other participant.

This feature is more of a "quality of life" decision, for users who desire conversation rather than confidentiality. By default, this feature *must* be disabled to prevent users from accidentally becoming visible when they join the network. This has no effect on a user's ability to see incoming chat requests – these must be visible regardless of whether the user is globally visible.

3.2.3 Group Features

Users *should* remain hidden in any group they join until they either reveal themselves explicitly or send a message to the group.

In practice, users may have good reason for being able to "lurk" in a chat group without their presence being advertised, but similarly they may have reasons for advertising their availability. The latter part of this requirement is implicit, as messages within the network are tagged with the nickname of the user which sent them, revealing their presence.

At group creation, a user *should* be able to specify whether or not a group allows for anonymous or hidden users.

In certain circumstances, users within a group need to be certain that they are the only participants observing what is being said as a matter of secrecy or privacy. If there are any unexpected eavesdroppers within a channel that can go unseen, then privacy for their discussion cannot be guaranteed.

3.2.4 Auxiliary Features

Users *could* style or format their messages, using MarkDown like syntax to create bolded or italic sections.

Plain text, while a useful messaging format, can often be seen as being a little bland for an instant messaging client. Modern programs such as Discord provide such functionality to allow users to better express a message's tone. This feature is not a requirement per-se, but would make the product more complete.

3.3 Elaboration on Requirements Arising From Threat Model

To protect users from external agents determined to discover which clients they connect to and converse with, onion routing is to be employed to obfuscate the route taken by any messages on the network where total privacy is desired. Since users all reside within the network, no traffic is exposed between end-points and destinations. This functions similarly to TOR hidden services and I2P's connection model, which only allow access from within the onion network. However, if a sufficiently high proportion of nodes within the network are acting maliciously or compromised then this property cannot be reliably maintained.

To keep users secure, a large focus of the design is minimising the amount of circumstantial information that can possibly be disclosed accidentally. To this end, nicknames in applications must merely be an ephemeral alias for a users current identity – a randomly generated public-private key pair, and users do not register for the service in an account based fashion. No other information from clients should be requested.

To reduce any risks presented by the potential compromise of a central server, location and connection information must be distributed across a peer based network. This works in tandem with ephemeral names – most information about users is hosted on their own client machines, with only a name stored elsewhere on the network – and since user account data does not need to be stored, this model becomes more viable.

To prevent name spoofing, it is required that only one user at a time may take a nickname – ensuring that a user can be reasonably certain that they are talking with the only client using a given name. Additionally, we

require defences in the event that a malicious actor chooses to overwrite nickname records or other protected data. To augment this, users must authenticate with directly connected clients using the public key attached to that node along with some certification step to determine whether some other malicious client is intercepting the data at link set-up.

It may be reasonable to allow users to reserve a given name for themselves, by giving records connecting keys and nicknames a TTL ("time-to-live") value and allowing users to save their public/private key pair locally. This allows certain users to make an explicit trade off – some persistence of their ID in exchange for introducing elements of risk elsewhere.

The choice to allow users to remain unidentified within groups for which they have access is a deliberate consideration – a client reading messages like this may still desire to remain truly anonymous, and it should be within their rights to do so. As other nodes on the network are likely to mirror the buffer of messages for a given group, such users would appear no different from mirror nodes.

Finally, data should ideally be duplicated across the network where possible, to reduce the impact of node crashes and potential DDoS attacks on users.

3.4 Derived Requirements for a Network Stack

From the above threat model, the requirements of the top-level application, and the requirements arising from the threat model, the required viable feature set for the network can be defined. The meanings of "Must", "Should", "Could", and "Would" apply as before to denote the value and importance of a particular feature.

The network stack *must* run in any modern browser supporting WebRTC.
Presently, WebRTC is the only vendor-neutral peer-to-peer API available natively within web browsers. Building a system without such access would require extensive relaying of all traffic across central servers. Additionally, WebRTC provides secure transmission between any two clients and allows for simple NAT traversal.

The network routing information *must* be fully decentralised.
Placing all of a client's trust in a single infrastructural server is dangerous. We must minimise the impact of a single point of failure, and reduce risk that a compromised network participant presents to other users by becoming a malicious actor.

The network *must* scale to arbitrary amounts of users, with little impact on lookup times.
If we desire a decentralised peer-to-peer routing scheme, then ideally the system must allow any active user to access or communicate with any other active user. As the network grows, naïve designs cause a heavy bandwidth cost as they require users to hold a large number of open connections. We need to minimise the impact of running the network stack on resource constrained devices to achieve the desired deployability.

Contact between any two users *should* support source verification as well as authentication.
We desire a strong link between someone's keys and their identity. For instance, if a user looks up the public key for any other node and a malicious actor serves them a *different* key then they must be able to determine that the key does not match that which was requested, and that in fact it belongs to a different user.

Identities on the network *must* be ephemeral.
The network identity generated for each user must have no connection to their *true identity* in any observable way, be this their IP address, MAC address of any of their network interfaces or any other piece of revealing data. If this can be inferred from a user's network identity, then the anonymity guarantees provided by onion routing fall apart.

Establishing new connections *should* not be mediated exclusively through a central server.

Ideally, further connections should be brokered through the peer-to-peer topology. In the event that such a signalling server were to be compromised or replaced for some nefarious purpose, then an attacker would be able to see records of connection data being exchanged between pairs of peers. If this occurs, then the connection data *must* be obfuscated, so that it cannot be read by an intermediary of any kind.

Cryptographic keys for any user *must* be globally accessible.

In the event that a user needs to verify the origin of a message (i.e. by checking a cryptographic signature) or that a user wishes to encrypt a message specifically for another user, then it must be easy to obtain the relevant public key to perform these operations.

The network *must* supply distributed file storage.

This allows users to place their cryptographic keys onto the network so that any other node can access them as needed. Additionally, this allows storage of files which connect cryptographically-derived identity to a temporary name tag for easy lookup, or to allow shared state between multiple users.

Stored files *must* have protection from unauthorised change.

A malicious actor could easily send commands to overwrite or modify a user's public key records, or any other crucial files they own. Nodes tasked with storing files must be able to recognise whether or not a change to a file's contents was requested by an owning user to guarantee file integrity and to prevent malicious modifications by adversaries.

The network stack *should* run on both server and client.

A server of some kind is required to coordinate WebRTC connection at all. If the server merely points to the network, then it over-privileges one or more nodes. This can have disastrous effects if all of these nodes disconnect and the server "loses" the network. By treating the server as another peer, users which become completely lost know how to return to the network at any time.

Developers *must* be able to build extend the stack to build additional transports and applications.

This is a key requirement to enable the development of onion routing – relay and route-building packets should be directed to hosts using the same rules as standard messages. The only difference is that these packets must be detected and handled differently.

Onion routes *should* be uni-directional.

In any drop-in-drop-out peer-to-peer network, a relatively high amount of churn is expected. In the event that any user which acts as a relay within an onion-route disconnects, the entire route is lost and must be re-negotiated. This would be time-consuming and incur additional recovery time for each side of the link. For a sufficiently large network, unidirectional links would be more likely to choose routes with no overlap, ensuring that if a relay fails or disconnects then the other side of a connection is unlikely to be affected. Additionally, this reduces the strain that each node in a relay must bear as it is likely to process less traffic.

The network *must* support usage of regular traffic alongside onion-routed traffic.

Application developers will not want to send all traffic with the security guarantees provided by onion routing – onion-routing systems are known to be expensive (in latency, bandwidth and computational cost). Allowing programmers the freedom to choose their guarantees allows for most networked traffic in a program's design to be carried far faster via standard means, imposing less strain on the collective network.

3.5 Summary

With reference to the desired feature-set of a security-focused messaging application and the capabilities of an attacker, the necessary features that the network stack must provide were obtained. Crucially, the Chord protocol provides much of what are requirements mandate. The core design will therefore have to adapt the

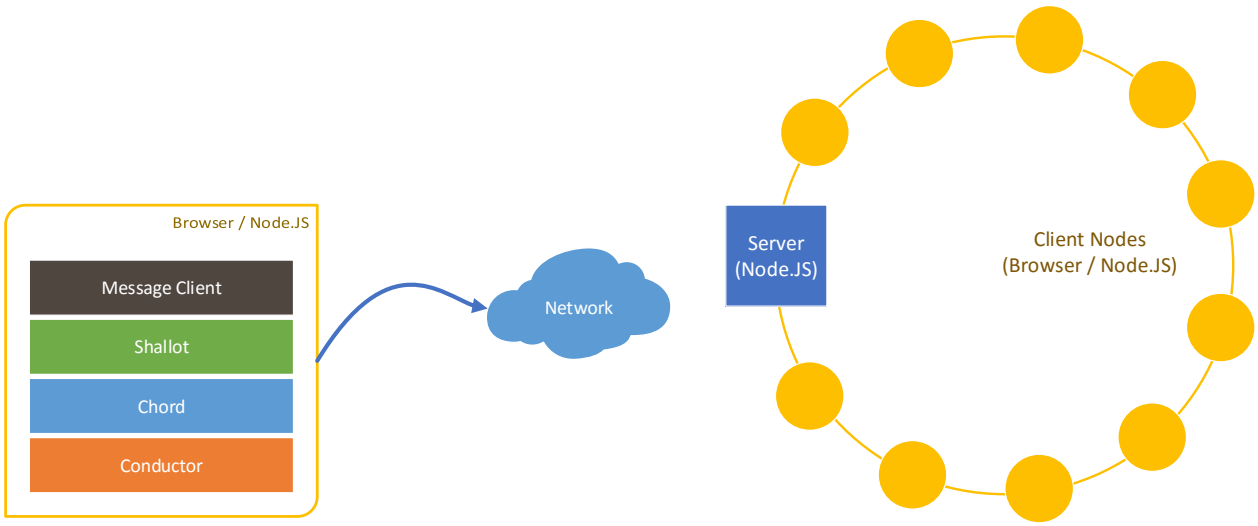
Chord protocol to include the concepts of file ownership and some means of constructing additional transports and services across the ring.

Chapter 4

Architecture and Design

This section presents an overview of the intended architecture for a messaging application, as well as an in-depth look at the design process, evolution, motivations and feature set of the project's various components. The design is broadly split into two sections – a network stack, which has been implemented, and a design for the application layer of the proposed messaging client. As the application layer has not been implemented, description of the design in these sections typically goes into less detail as the designs have not had to be adapted during development.

4.1 Overview



(a) Structure and environment of a client application.

(b) High level network structure.

Figure 4.1: *Simplified system structure at client and server level.*

At a high level, the system is composed of a stack of JavaScript modules within a browser which connect outward to some peer-to-peer network to run the message client application as in **Figure 4.1a**. Due to the requirements imposed on the design by WebRTC, a server node of some kind is required to allow browsers to discover and connect to other peers – **Figure 4.1b** displays this.

From the initial division of the design into a network stack and application layer, we may subdivide into

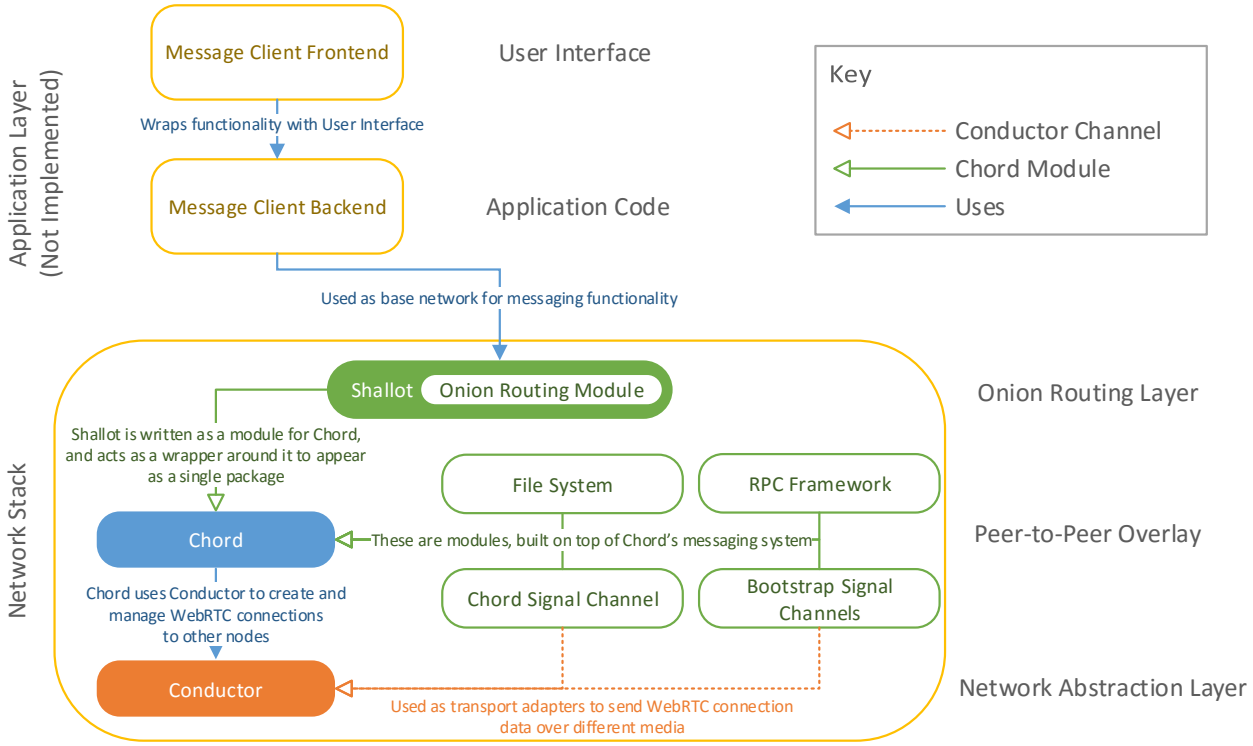


Figure 4.2: *Architectural diagram for the proposed messaging application, with emphasis on the network layer.*

further libraries and components. **Figure 4.2** shows this breakdown in detail, splitting the application layer into a user interface and application logic layer and displaying the core libraries within the network stack (shown in solid colour). Of particular interest are their interdependencies and relationships with one another - especially within the core networking layers.

Conductor's purpose is to completely wrap, manage and control all access to WebRTC data channels within the system, ensuring that creation of new connections is simple. Conductor is used exclusively by Chord to create new connections and to connect to the Chord network – initially to a super peer hosted on a server, before it becomes able to negotiate connections directly through Chord by treating this super peer as a proxy. Crucially, these represent *two different ways of exchanging connection data* which must be handled and considered differently. Chord provides Conductor with an implementation of a *Conductor channel* for each transport it intends to use for signalling – these contain instructions on how data should be transmitted, parsed and how the system should be notified that data has been received over a given transport.

Chord is responsible for providing the peer-to-peer overlay system, ensuring that any connected node can reach any other node in a decentralised manner in a very small amount of operations compared to the network size. Additionally, key features that it exposes to higher levels are *distributed file storage* and *public key lookup*, which are crucial for building distributed systems which require security. Chord also provides *module registration* and *messaging* capabilities to application developers, allowing custom services to exploit the fast lookups and reliability that it provides to build additional transports and services.

Shallot is a module built on top of Chord, using the features that it provides to implement onion-routing without reference to a central directory server. In particular, it makes use of Chord's capabilities to identify nodes which are responsible for IDs, global file lookups to obtain nodes' public keys and its messaging functionality to tunnel data along several nodes. Shallot has no awareness of any level beneath the Chord overlay network.

The ***Message Client Backend*** is the secure messaging client itself, designed to be implemented as another module built on top of Chord; with full knowledge of all the functionality it provides as well as that which is provided by the Shallot onion-routing module. Similarly, Shallot has no knowledge of the network stack below the Chord network.

The ***Message Client Frontend*** is intended to act as a simple user interface over all functions and capabilities supplied by the message client backend. As such, it has no knowledge about any level lower than this and interacts purely with the backend.

4.2 Network Abstraction Layer - Conductor

Conductor is designed to drastically simplify connection over WebRTC for higher level application and network layers by providing a transport-agnostic wrapper around the process of creating connections and data channels. Additionally, it is designed to be cross-platform by allowing for injection of WebRTC implementations in non-browser environments. The module is designed to exploit the core set of actions needed to transport data over a network by defining an interface and a set of primitive actions that allow an implementation to deliver SDPs and ICE data given a name or id for the desired partner. In addition, Conductor wraps the returned connections to simplify the process of data transmission, addition of further data channels and to simplify WebRTC's unusual disconnection semantics.

The WebRTC protocol requires connection negotiation to occur out-of-band, via the use of some other transport medium to exchange the information needed to open a new peer-to-peer connection. As a result, the API provided to developers is quite in-depth compared to simpler models of connection such as WebSockets. Basic usage of WebRTC demands manual handling of SDP and ICE creation and parsing, often introducing a vast amount of asynchronous set-up code that is virtually identical each time regardless of the transport medium chosen. One of the key motivations is removing this boilerplate code, and inventing a simpler, more modular abstraction over the API.

The design of higher levels within the network stack strongly emphasises the importance of being able to build connections in a variety of different ways while maintaining consistent naming and lookup across transport protocols. Conductor introduces the concept of ***Conductor channels*** - these are objects which implement three key actions:

1. `.send(id, type, data)` – transmission of data of a known type to a peer, according to its ID.
2. `.onmessage(msg)` – reception of data, deciphering the return type, content and sender's ID.
3. Notification of Conductor upon reception of connection data, as well as the ability to parse this data.

This abstraction allows for higher levels to provide code which implements these simple operations, leaving Conductor to handle and coordinate the data exchange as needed. These channels are extremely flexible - two peers could theoretically exchange data to one another using completely different protocols for request and reply or even chain a series of channels together locally. Conductor additionally exposes some methods to channels so that they can rename connections, or reject them upon failure. As a design feature, protection of the sensitive SDP and ICE candidate data is left to the discretion of the channel implementer – it is entirely possible that transmission may occur over a known secure medium, such as TLS.

Most significantly, this abstraction now allows creation and use of a new connection to be expressed in a single line, `conductor.connectTo("exampleID").then(conn => /*...*/);`. This simplifies the connection process significantly – one of my early experiments uses almost 200 lines of code to open a single

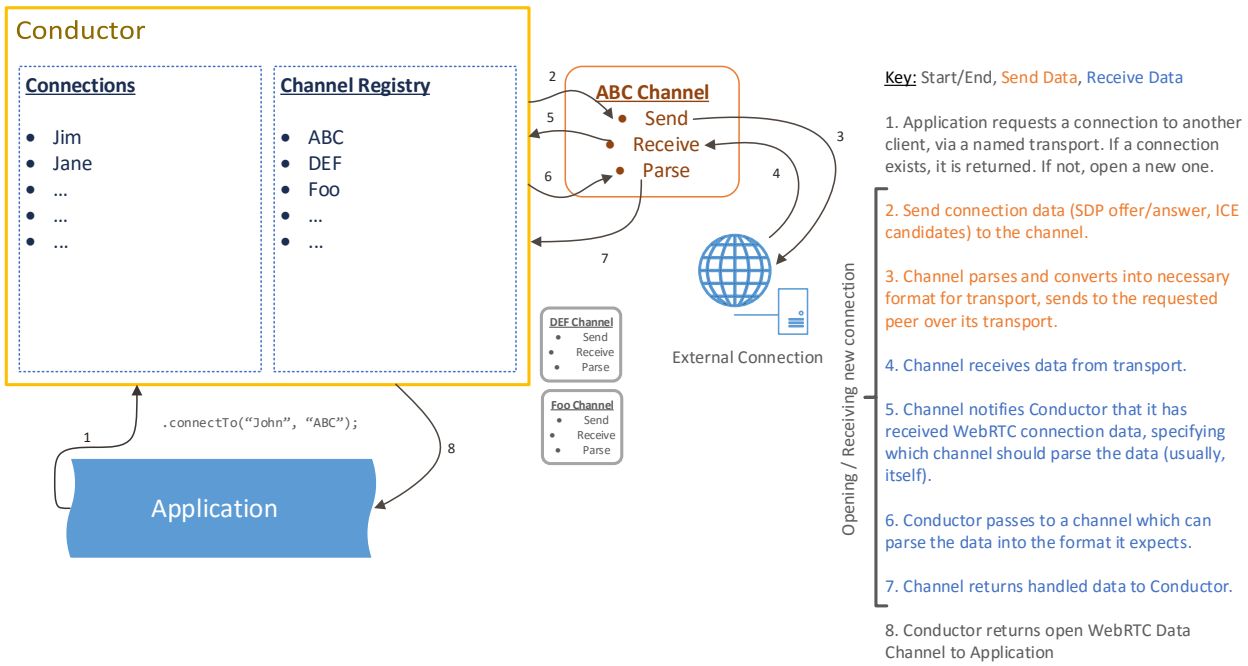


Figure 4.3: *Graphical overview of operation of Conductor channels, as well as its role as a central "directory" of all open connections. This covers direct request of a new connection – handlers may be used to automatically perform actions when a connection is* ***received*** *as well.*

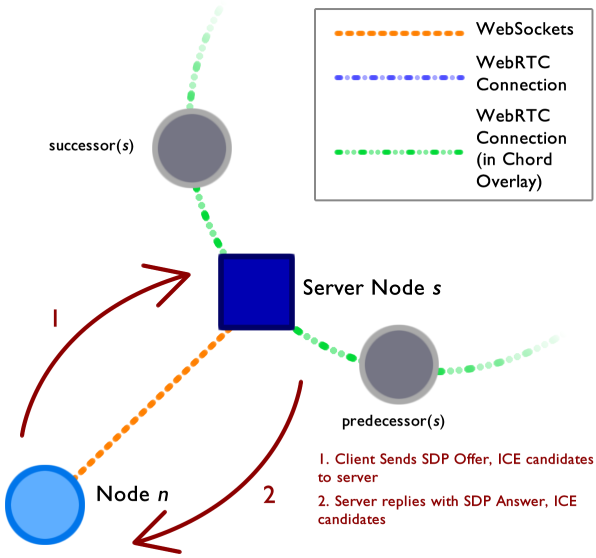
connection using WebSockets over a relay server as a signalling channel, with no encryption or handshakes. The channel implementations for each side take slightly less code to achieve the same effect *with these additional features* – although Conductor does take more lines of code overall, as the amount of signal channels to be handled increases it should become more efficient due to its generalised design.

WebRTC's connection state semantics are rather different from a standard model where disconnection is typically considered to be a terminal state. Considering the state machine on the WebRTC specification [8], it can be seen that it is possible to return to a live state from disconnection or failure. This model makes the most sense for VoIP and video conferencing, but when managing an overlay network topology a definitive and swift answer is needed when determining whether a link is still open or if a node is still reachable. As a result of this, Conductor treats failure states as terminal, forcibly ending connections to simplify WebRTC's richer failure semantics into a more traditional form.

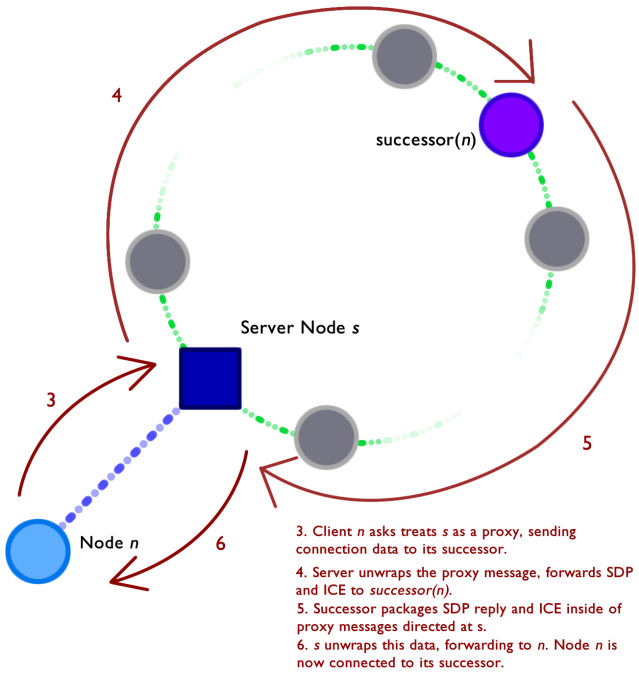
4.3 Peer-to-Peer Layer - Chord

A modified variant of the Chord protocol is used as the underlying peer-to-peer topology for the network stack. The standard design is now divided into *messages* and *modules* – messages contain fields denoting which module is responsible for interpreting them. This introduces a greater degree of extensibility by allowing application developers to design and implement their own protocols on top of Chord. Additionally, *self-certifying identifiers* are utilised to allow for simple source verification and authentication without use of a complicated public key infrastructure. Chord was chosen for a multitude of reasons. It provides the scalability and fast access to any other node within the system which we require, it's proven to be extremely reliable [14], there are many reference implementations in various languages and it provides the distributed file storage which will enable the application design.

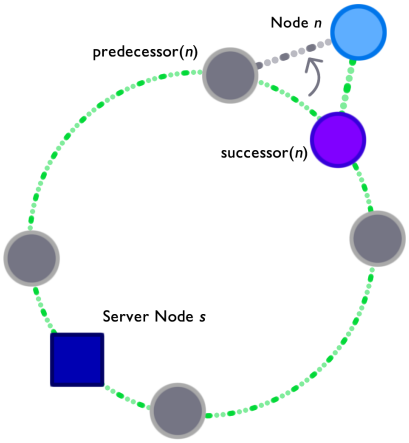
While the Chord protocol acts as the basis for an extremely resilient and powerful overlay network, certain elements of its design make it extremely unsuited for modern application design. Traditionally, Chord is designed so that nodes notify one another about application-level events by placing (and removing) files from the system. This model works well for certain classes of application, such as work division in a distributed computation system or within a content delivery network, but for more traditional applications we require explicit, targeted communication in addition to file storage. By redesigning Chord to be based upon a message-driven model, I manage to fully separate the concepts of permanent storage and ephemeral communication from one another, pushing the former into a simple self-contained module and exposing the latter to make the system more extensible as described above.



(a) The server node runs a WebSockets server, which it uses to exchange connection data with new clients.



(b) The client node asks the server for its successor's ID, and proxies the needed connection data across the known server node.



(c) The client node has found its place in the network after connecting to its successor, its predecessor will be notified of its existence and establish a connection shortly. More connections may be opened via Chord messages.

Figure 4.4: *Overview of server and client roles in the system during entry.*

Because WebRTC requires the use of a server to allow signalling between clients, the design necessitates a server which is in some way aware of the existence of the peer-to-peer topology. This presents two choices: the server either maintains a constant connection to one or more known peers and simply acts as a relay, or the server node is a peer itself, hosting a signalling server that it can use to create external connections. As the former option is particularly vulnerable to changes in the network topology such as disconnects, my approach treats the server node as a nominated peer within the system. **Figure 4.4** shows how this works in practice, with the server node acting as an "anchor" for the Chord peer-to-peer network.

Remote calls used by Chord to manage the topology *must* be carefully handled – if too much privilege or power is extended to other nodes then an attacker could easily exploit this to disrupt the flow of the network and deny service to any number of clients. To this end, the stability-oriented variants of Chord's management methods are used [22, p. 155]. These procedures act exclusively through information lookup and periodic notification of other nodes, where topology changes are always verified by the notified node. This allows for the removal of many of the other operations which do allow direct, unchecked control of other nodes' routing information; thereby increasing the defensiveness of the protocol.

4.3.1 Identity Allocation

This implementation of Chord makes use of *self-certifying identifiers*, a concept introduced in "Routing on Flat Labels" [4]. The concept allows us to meet our initial requirement of providing source verification and authentication – by having a node's identifier be derived through a non-reversible computation on its public key, we can map any public key in the system against the node to which it belongs. On creation, each node generates a public-private key-pair for use with any valid asymmetric cryptographic algorithm. A node then obtains its identity by calculating the hash of its public key using a strong cryptographic hash function.

Once a node has generated an identity it then stores its public key in a file within the network, which can be accessed by other nodes who wish to contact it securely. The node then routinely checks that the file remains accessible, and that it's contents are correct. Crucially, the file's name *is the node's identifier* – this allows other nodes within the system to obtain the key by performing a lookup for a file named after their intended correspondent's ID. This means that a node does not store its own public key within its region of the file system (it *does*, however, remember it internally). In the event that a key-file is lost or is somehow modified, a retrieving node will notice its absence or that the hash of the contained file does not match the file's name. The node may then attempt to ask its intended correspondent directly for its key, with the above properties acting as an integrity check on the transmitted key. Access through the file system is preferred where possible, as it reduces the strain on the intended correspondent. Similarly, this scheme allows source verification in combination with digital signatures – a module needs only look up the purported sender's key to determine if a message's origin has not been substituted or spoofed. Crucially, this introduces the required source verification and authentication *without* the use of a complex public key infrastructure as in [3].

4.3.2 Message and Module Backend

A combination of message passing between nodes with an extensible module and delegation system forms the basis for this peer-to-peer network, and proves very effective in practice. A state machine is utilised to determine the connection state of Chord, which in turn informs the message routing rules – enabling smart proxying and allowing message transmission and reply across intermediate network states to be easily performed so long at least one other node is known. The network topology used is a variation of Chord, designed to be more suited to development and implementation of more traditional applications by moving to a message- and module-based approach. Additionally, the protocol must be adapted for usage in environments where there are strong restrictions

on connectivity – in particular, WebRTC, where it is essential to already have some means of communicating with another client before opening a direct link.

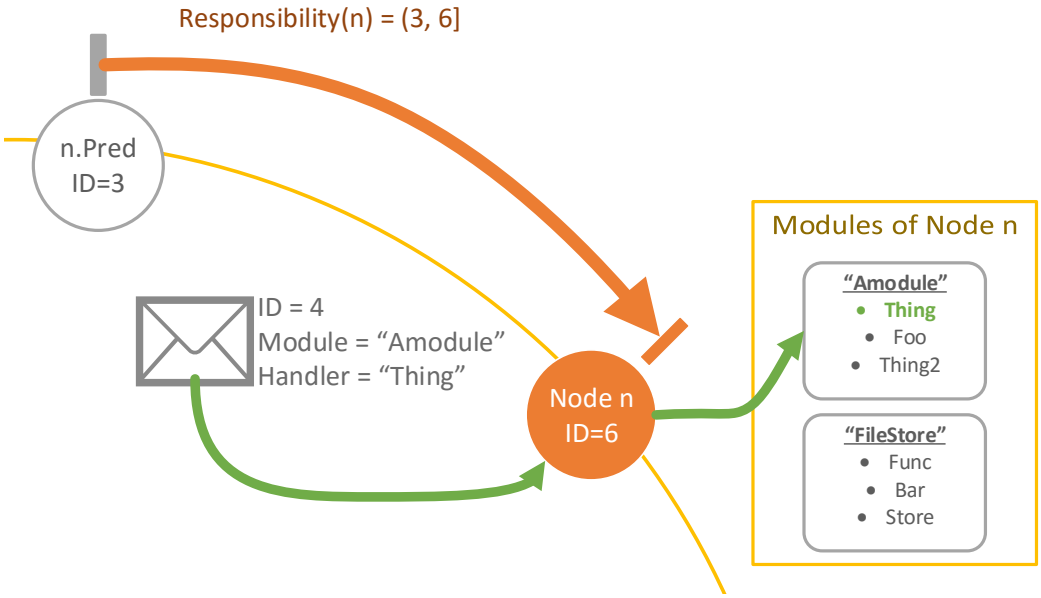


Figure 4.5: *Diagram of the message and module system in action. Nodes are responsible for handling all messages whose IDs are within their area of responsibility. Messages are parsed based on "module" and "handler" fields upon reaching their destination. In the above, Node n consumes the message as 4 falls within its range of responsibility – it then selects the correct module and function from its internal registry to act on the message.*

Initially, the design was built to send only simple JSON (JavaScript Object Notation) objects between nodes, containing a message field and destination. The simplest test of operation was performed; nodes would consume and print the output of any message whose ID matched their own. While this helped in the initial development of Chord, as the design progressed it became clear that this model was too simple, and would not suffice with regard to the network's requirements. One of the most important requirements of the peer-to-peer topology was to provide extensibility, as well as different treatment of packet classes. For instance, the system needed a clear separation between Chord's control traffic and any other arbitrary use case – despite both being governed by the same routing and handling rules.

The design of Chord was then divided into *modules*: objects with a known name to match packets against, and which provide an interface that allows Chord to `.delegate(...)` handling of (and reaction to) incoming data. Transferred packets would be augmented with "module" and "handler" fields, which would be used to indicate the intended module and specified action within that module. Modules would be registered through a central manager within Chord, and this functionality would be exposed to higher level application developers. Completion of the basic module system then allowed classification (and sub-classification) of packet varieties, enabling the design and implementation of the Remote Procedure Call framework as well as the initial implementation of Chord's routing algorithms.

Similarly to how standard Chord requires that a node n is responsible for all files inside the identity region $R = (predecessor(n), n]$, the base concept of the messaging backend is that a node is responsible for consuming all messages which are directed anywhere within R. This is shown by **Figure 4.5**. Message routing rules during this stage of development were extremely simple: if a connection to the destination exists, then forward the packet immediately. If the message falls within R, then direct it to the proper module – else, forward to the closest preceding finger for the message's destination ID. During connection, these rules did not account for the case where the successor or predecessor had not yet been identified – special handling was needed in this cases for the formative stages of the network. When trying to develop fault-tolerance and recovery from disconnections,

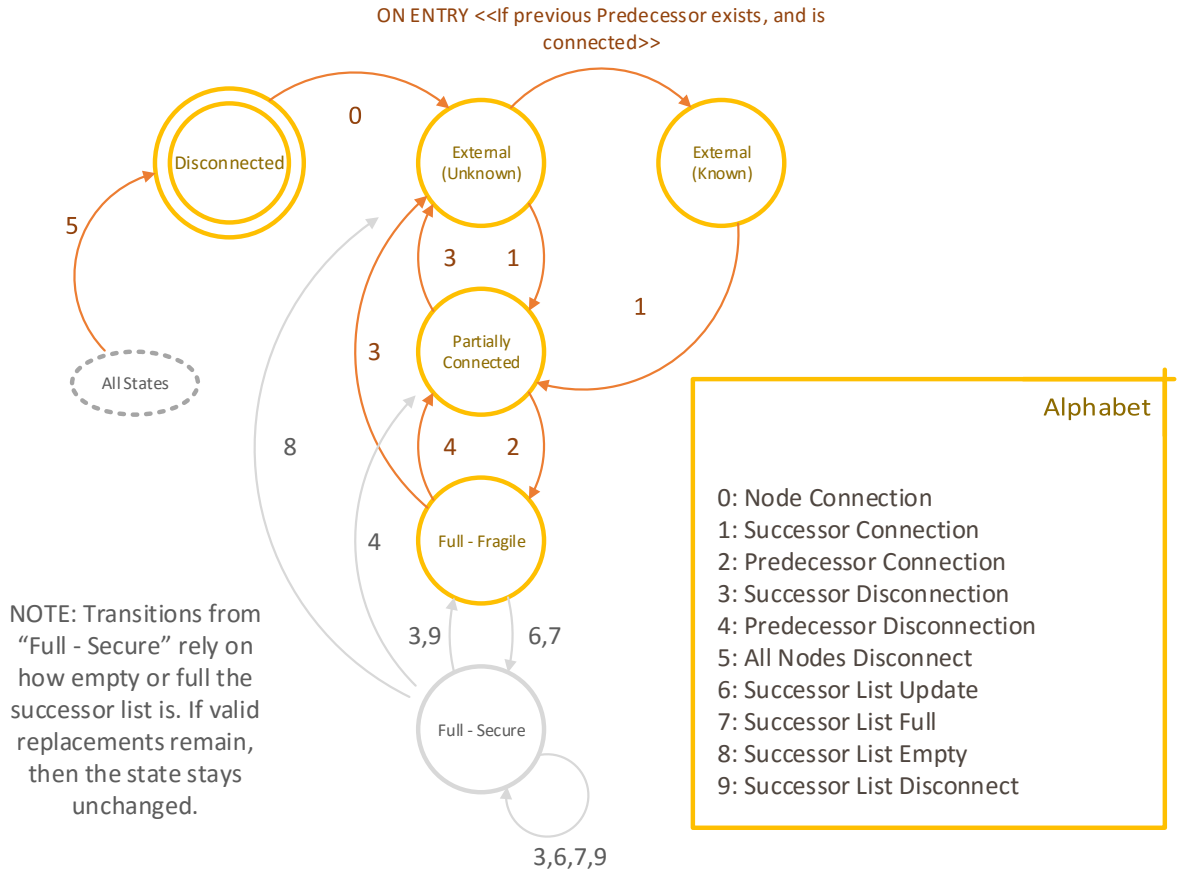


Figure 4.6: *Finite State Machine governing any node's state of connectivity. Although backup successor lists (the state "Full-Secure") are modelled for as suggested in [22] to achieve a more robust system, they are not implemented in the final design.*

these rules became far too complex to maintain; it was determined that modelling the system as a finite state machine was necessary to make routing feasible in the face of high network churn. With reference to the Chord specification as well as consideration of WebRTC's requirements, the system was modelled as in **Figure 4.6**. Each state then corresponds to a choice of node to pass a message over, as displayed in **Algorithm 1**.

In addition to its use for routing, the state machine is also used to enable reconnection to the network. While a partially connected node (i.e. one which is connected to its successor) is guaranteed to be brought into the network to a full state by Chord's routing algorithms, external nodes are aware of nodes within the network but are not guaranteed to be known themselves. Nodes which are truly external (and are totally unknown to nodes within the ring) must go through the initial connection procedure once more, but nodes which believe themselves to be known have a high likelihood that another node "remembers" them and can bring them back to full connectivity. In the event that this does not occur, the node waits for some period of time and rejoins through the initial connection process. The *"Full - Secure"* state represents one of the proposed enhancements by Stoica et al., where nodes would obtain lists of (and connections to) the next available successors in the event that their true successor fails; my present design neither gathers nor acts upon this information.

The use of WebRTC imposes strong connectivity constraints within the system – in essence, a node requires connections to at least one other node to create more connections. Although this has less impact when simply forwarding a message, great care must be taken when another node is expected to be able to reply. This is especially significant when using Chord itself as a signalling channel – messaging another node such that a reply

can be routed back relies on the knowledge of how how best to contact them from the current state. To handle this initially, the chord signal channel routed every piece of connection data via a proxy node, chosen according to the current state. Although this approach was admittedly extreme, it was the basis for the next iteration of the message backend's design.

Data: Own ID "n", Message Content, Message Destination
Result: Eventual delivery to intended recipient

```
1  if Direct Connection to Destination then
2      send message content to Destination;
3  else if Destination = n or State = Disconnected then
4      // Destination is a direct match, or n is effectively a 1-node ring.
5      direct message content to appropriate module;
6  switch State do
7      case External
8          // We are connected to at least one node.
9          // Assume that it is in the Chord ring.
10         proxy message content over first available connected node;
11     end
12     case Partial
13         // We are connected to our known successor.
14         // Assume again that it is in the Chord ring.
15         proxy message content over successor(n);
16     end
17     case Full - *
18         if Destination ∈ (predecessor(n), n] then
19             // Only state where we can calculate the region R.
20             direct message content to appropriate module;
21         else
22             // Choose the best finger to send the message along.
23             send message content to closestPrecedingFinger(Destination);
24         end
25     end
26 endsw
```

Algorithm 1: *Final message routing rules.*

The final design of the message core builds the capability for message proxying into Chord itself as part of the message design, as well as allowing for packet validation across multiple packet formats and hop limits on messages. The format, roughly, is a two byte version code, where the standard format operates using JSON (**Figure 4.7**) – full discussion of the packet format itself is provided in **Appendix B**. This complete format redesign required all existing modules to be adapted, but massively simplified the semantics for ignoring and replying to messages – particularly where proxying is concerned. It is worth noting that whenever a message needs to be proxied, the likelihood that the message is lost sharply increases. In essence, proxying is a last resort "heuristic" for directing a message to the rest of the network – it is entirely likely that Chord could make the wrong choice, *particularly when in the "External" state*.

```
00{
    "s": "...",
    "d": "...",
    "...": "..."
}
```

Figure 4.7: *Sample of Chord's packet format.*

The choice to allow multiple packet formats is deliberate – while JSON is easily readable for both humans

and machines, it is relatively space expensive compared to raw binary formats. Having the capability to introduce additional packet versions into the system allows for not only different formatting to be handled, but potentially different classes of packet altogether. Some of the applications of this are discussed in **Section 8.2**.

4.3.3 Remote Procedure Call (RPC) Framework

As JavaScript lacks the RPC functionality typically built into more established languages such as Java, the design requires that we build one from scratch to utilise the Chord message transport. Although initially this functionality worked in the simplest possible way, it has been extended and redesigned with multiple retries, effective error handling, configurable timeouts and at-most-once semantics. These features help to make the system (and all modules built on top of the framework) more robust and fault-tolerant.

Modules make use of the framework by subclassing `RemoteCallable`, giving them access to the ability to make, answer and fail any remote calls directed towards the module. Every call made from the framework is given an integer ID, and two functions are stored locally which resolve or reject the call – representing receipt of an answer or error, respectively. In the interim, the caller is handed a `new Promise()`, which is a JavaScript language construct representing an asynchronous action. The message's handler is set as the name of the method being called – the module on the callee's node is then responsible for responding to this as it sees fit, but it *must* reply with an error or answer packet. When the caller receives a packet with the handler "answer" or "error", it looks inside of its internal storage object and resolves or rejects the initial `Promise` with the given answer or error.

To handle errors and packet loss, timeouts are used so that a call is said to have failed if no answer is received within the allotted time. A call may then be retried, or the call can fail altogether if the module designer so wishes. However, consider both cases where a message is lost en-route to the callee, and where the answer is lost while being sent back to the caller – in the first instance, retrying the call is likely harmless, yet in the second case this call could affect state or return a different answer if retried! As a countermeasure, nodes cache all answers they return for a configurable amount of time. Because of this, if a call is retried at any stage then the cached answer can be immediately returned, requiring no further computation and preventing unintentional mutation of state.

From a security perspective, developers are free to expose only a subset of a module's functionality remotely if they wish to reduce the attack surface exposed to adversaries. Ideally however, the application developer should either make remotely available procedures idempotent, or minimise any unverified mutation of state where possible.

4.3.4 Bootstrap Channels

As demonstrated in **Figure 4.4a**, initial connection to the network is performed by exchange of data over WebSockets, a datagram focused extension of HTTP. Rather than using the WebSockets server as a simple relay, the server is treated like any other node and the new client is able to join the network using standard Chord algorithms.

To do this, the server maintains a reusable Conductor Channel around a WebSockets server, while clients create a throwaway channel which acts as a WebSockets client for signalling. For simplicity, clients create a new channel whenever they desire to connect to a server. On creation, a client channel connects to the WebSockets server, and exchanges public keys with the server, from which the two can derive one another's identities. The server then encrypts an asymmetric cryptography key using the client's public key and transmits this, finishing initialisation. The server attaches this data to the record of the WebSockets connection that the client opened. The channel prevents Conductor from transmitting any data until set-up is complete.

Use of an asymmetric cryptographic key is particularly important here. Establishing a secure WebSockets server requires keys and certificates, and a simple server may not have access to this. As a result, all data sent between client and server is sent in the clear, and *must* be encrypted – this becomes an even greater requirement with regard to the sensitivity of SDP and ICE data. Exchange of SDP and ICE data over the channel may now occur, using this shared secret key for encryption and decryption. Finally, the client has a secure WebRTC connection to the node hosted by the server.

4.3.5 WebRTC Connection Negotiation over Chord

Crucially, a node needs only to know about one node within the network to establish further connectivity. Taking inspiration from Vogt, Werner, and Schmidt in [28], we may treat the Chord network itself as a Conductor Channel: all further signalling negotiation of connections is performed exclusively through the Chord network. However, a newly joining client remains outside of the Chord ring, and is only known by a single node – standard messaging rules will completely fail to direct any replies to the new client. To resolve this, proxying was introduced to the design, as demonstrated in **Figure 4.4b**.

Although WebRTC Data Channels are secure on a hop-by-hop basis due to the use of DTLS, the data must pass over several nodes to reach the intended recipient. Data routed across an adversary could be read or modified accordingly. As a result, encryption of SDP data - again - *must* be employed to prevent intermediate nodes from interpreting the sensitive data. Before connecting to one another, nodes perform a handshake across the network, exchanging keys and IDs as in the bootstrap process. As the initiator may not know the correct ID of the node it wishes to connect to, this handshake is particularly important for confirming the identity of an intended partner. While the handshake is ongoing, the module queues up all messages which Conductor tries to send, and acts upon them once the handshake completes.

Initially, all communication through this channel was routed across a proxy chosen according to the current state, through similar rules to those in **Algorithm 1** (except that in a state of full connection to the ring, the node's successor is chosen). Although this scheme worked, needlessly proxying all conductor traffic proved to be a waste of resources and time for nodes which were already *in* the network.

To alleviate this issue, proxying rules and capabilities were moved from this module into the messaging layer itself, as discussed in **Section 4.3.2**. Although the basis for this choice was sound, it revealed an underlying flaw in the implementation and design of this signalling channel. Consider the situation in **Figure 4.4c** where a new node n has successfully joined on to its successor m. The node which was previously m's predecessor, denoted p, learns from m that n is its new successor, in turn this node tries to establish a connection to n to fully bring it into the network. However, p has no idea that n lies outside of the network and does not send any of the signalling traffic through a proxy. This data reaches n successfully, but when n tries to reply to p using the standard rules it finds (without fail) that p's ID falls within its region of responsibility and consumes the reply.

Two solutions were considered for this. p could either proxy signalling traffic across its old successor m in the event a new successor is discovered, or n's module would simply ignore all non handshake initialisation traffic if the ID was not an identical match. In practice the second approach was taken – it was not only a far simpler implementation, but it required no changes to the routing rules or for the module to remember any data about the network's topology. Once this change was made to the design, Chord could freely establish connections using any known node within the network, massively reducing reliance on a server for signalling.

4.3.6 File Storage

In contrast with the standard design of Chord, we move the distributed file system from the core of the system's event model to simply being a service built on top of the messaging layer. Additionally, since the file system is no

longer overloaded to represent both storage *and* notification, we are now able to further specialise the module by introducing the concepts of authentication and ownership of files. Previously, the mixed semantics would have made an authentication scheme troublesome, particularly when multiple nodes attempt to direct notifications to a single target node using the same file key.

Initially, this module operated in a very simple, trusting way. When asked to store any file, the responsible node would simply overwrite the contents of its internal storage at that key's location! This has risks for protection of public keys within the network – it was crucial that the design was adapted to prevent malicious users from overwriting important state used by other nodes. To combat this vulnerability, it is essential to introduce the concept of *file ownership*.

Broadly, what we desire most from our implementation of ownership is to ensure that only the node which first places a file onto the network has the right to edit, replace or explicitly delete the file matching that key. Transference of ownership to another node is a secondary goal. The simplest way to do this is by making use of some *shared secret*, such as a random string, between the owner and the storage location. This shared secret can then be treated as a cryptographic key for a symmetric algorithm such as AES, and can be used to verify the origin of updates. Additionally, this allows for transference of ownership by communicating the public key to another node. We can now define the procedure for storing files:

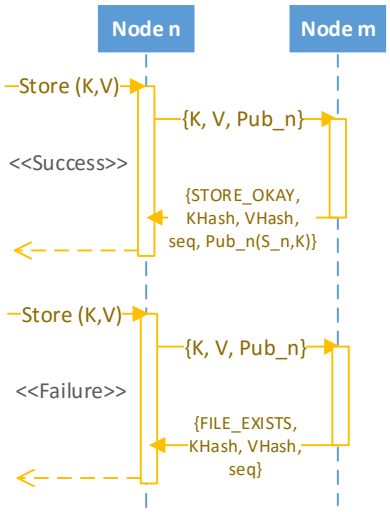


Figure 4.8: *Sequence diagram for storing a file.*

To place a file onto the network, a node n is responsible for sending the key-value pair representing the file name and data, K and V respectively, as well as its own public key Pub_n to the node responsible for the ID $Hash(K)$, defining a node's responsibility identically to within the messaging backend. We shall denote the intended recipient as node m. When m receives a remote call requesting storage of a file, it recomputes the hash of the file's key – if the message was routed to it erroneously (for instance, if an adversary intends to overwhelm m), then it forwards the request through the network instead of parsing it. Node m then checks its internal storage tables – if an entry already exists for $Hash(K)$ then the initial call is replied to with an error code. Crucially, completely overwriting a file is not allowed at this stage – it is expected that a service which owns a file would be aware of this fact and only choose to update the file. If no file exists, then m stores the K, V pair locally and generates a secret 16-byte ownership key, $S_{n,K}$ – i.e. n's secret key for the file K. This is proof for m that n is indeed the owner of the new file in any future updates. Node m then answers n's call with an object including a success code, the file's current hash ($Hash(V)$), the initial sequence number (0) and encrypts this ownership key with n's public key ($Pub_n(S_{n,K})$). n is then free to decrypt the secret ownership key using its private key, and then stores this value for future updates.

However, there is a high likelihood that a client which owns many files across the network could disconnect at any time. For long running networks with high amounts of churn and file ownership per node, this would eventually lead to a high quantity of "dead files" where their owners have long since left – in turn preventing reuse of their owned file keys. n responds to a successful file store by running a periodic task, sending a keep-alive message to the current owner to ensure that the file remains accessible. Files on the network are held for a configurable time window – if the file's host receives no keep-alives during this time period, then the file is removed from storage. This is additionally employed as a counter measure for nodes which desire to own a vast selection of useless files to disrupt the network and impose high storage costs, by requiring that they maintain a constant stream of network traffic in exchange.

To update a file, we require a secure design which utilises the secret ownership key to prove that an update did, in fact, originate from a trusted node. At first, the design had the file owner send a random string s along with its encrypted form $S_{n,K}(s)$ – however this proved wholly insufficient after some thought due to its vulnerability to *replay attacks*; an attacker could send a previously seen file update command to reset the file to an earlier state. Two solutions were proposed to this: 1) have the storage node send a challenge to any node which wishes to perform an update, or 2) maintain an internal sequence counter which increments on each side per update, where the storing node rejects any updates with an incorrect sequence. Option 2 was chosen, as I came to conclude that challenges being sent across the network would increase the time taken and complexity of updating a file. To update a file, the following procedure is implemented:

Node n wishes to update a file, $\{K, V_{old}\} \rightarrow \{K, V_{new}\}$. As proof of its identity, it produces the string $proof = S_{n,K}(Hash(V_{old}) + sequence)$ (where $+$ denotes concatenation) by using $S_{n,K}$ as a symmetric encryption key. n then directs a `.update(...)` call, sending $\{K, V_{new}, proof\}$ along with the tag and initialisation vector used for encryption to the node responsible for $Hash(K)$ – we shall denote this m as before. Upon receipt of this call, m decrypts the $proof$ using the stored secret key and sent initialisation vector, before verifying against the included tag. If the unpacked hash matches that generated by the stored file V_{old} and $sequence \geq sequence_m$, m's local copy of the sequence number, then the file is stored locally and m updates its sequence for the file, $sequence_m = sequence + 1$. n is sent a success code, along with the hash of the new file contents and the updated value of $sequence_m$. If decryption fails or either piece of data was incorrect then a generic error code is returned, as well as the hash of the stored file and the current value of $sequence_m$. Note that this does not need to be kept secret – this takes the place of a traditional challenge. The main advantage gained by this approach is that n need only be reminded of either value if it is not the only owner of a file, reducing ring traffic and the amount of time needed to exchange challenge information while still defending against replay attacks.

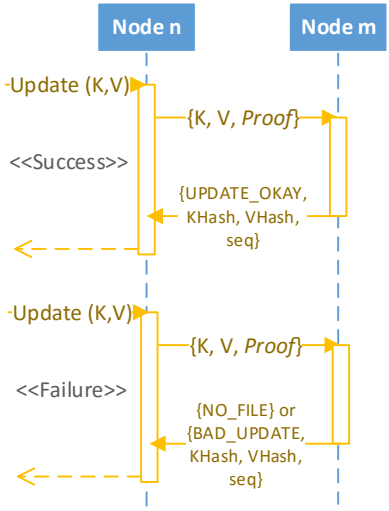


Figure 4.9: *Sequence diagram for updating a file.*

As the network grows and each node's region of responsibility changes, it becomes important for nodes to redistribute files which they hold – not only to alleviate their burden of storage, but primarily so that each stored file can still be located. To do this, nodes check to see which files they hold which fall outside of their responsibility both as a slow periodic event and in response to changes to their predecessor within the network topology. This follows from the definition of a node n's responsibility, $R_n = (predecessor_n, n]$ – it is clear that the only way to change a node's responsibility is to change its predecessor node. In either of these cases, nodes scan over the IDs of all files in their storage. If they are no longer the designated storage location of the object, then they identify the node who *does* have responsibility of the item and negotiate a transfer of the object, the current sequence number and the secret ownership key. This can be performed securely by obtaining the destination node's public key and encrypting the data so that only they can read it.

In its current state, the file ownership and authentication schema is vulnerable to several attacks – these are discussed in greater detail in **Section 6.2.1**. This design is preliminary – future extensions and modifications that would alleviate these problems are discussed in **Section 8.2** as well as alongside the attacks themselves.

4.3.7 Improvements over Chord

In addition to the domain-specific observations and modifications to Chord presented here, some optimisations to the algorithms present in the protocol itself are proposed.

The stability oriented algorithm for updating a finger table [22, p. 155] can be enhanced to make finger pointers more accurate in fewer lookups, by recognising better approximations as they become available. For a given node n, when periodically updating a random entry i within the finger table, if the discovered pointer p is a better fit than $finger[i]$ (i.e., $p \in (finger[i], n + 2^i]$) then the system sets $finger[i] = p$. The system then increments i, and repeats until p is no longer a better approximation for $finger[i]$ or the end of the table is reached. This process may also be conducted if, say, n is connected to by p as one of *its* fingers. Although p may not be the canonical "best fit" for successive finger table entries, it is indeed a closer approximation to the theoretical best finger pointer, $n + 2^i$. As a result, choice of the next hop for message delivery is still guaranteed correct and we are likely to see any messages to further IDs delivered in fewer hops.

To maintain correct routing, an additional invariant upon the finger table is maintained – this proves particularly important when transitioning between different levels of connectivity, such as after responding to a predecessor disconnection event. As message routing relies on the `.closestPrecedingFinger(...)` algorithm, for a node n we require that every finger in the system falls within the region $(n, predecessor(n)]$. This algorithm is found within the Chord paper [22]. When any change occurs to the predecessor, Chord scans the finger table from the end, setting any fingers outside of the above region to equal $predecessor(n)$. Although other implementations may not rely on this invariant holding true to operate correctly, the check is inexpensive to perform, and needs only to occur if n's predecessor disconnects or is replaced.

4.4 Onion Routing Layer - Shallot

Shallot provides onion routing built on top of Chord's extensible message delivery by adopting a hybrid model between Tor and I2P's protocols. Route creation and packet transmission are performed using algorithms almost identical to those utilised by Tor for route generation and data transmission. This maintains perfect forward secrecy by mimicking Tor's well-established design, with the only significant divergences lying in circuit ID obfuscation and that all symmetric keys are generated by the source of a link rather than by successive handshakes.

As a module for Chord, Shallot's design takes a lot of influence from the design of I2P. Unidirectional links are adopted using a circuit-switched approach for multiple reasons. Primarily, this approach embodies simplicity; making it particularly suited for proving onion routing's feasibility in this environment. More importantly, it lends the system a far higher degree of fault tolerance. As the amount of nodes in the network grows sufficiently large, it becomes increasingly unlikely that each directional link between a pair of endpoints share any relay nodes. Following from this, the disconnection of a relay node is less likely to close both routes simultaneously.

Drawing further inspiration from I2P's design, Shallot provides datagram-focused service to applications rather than Tor's stream-based approach. This makes usage in higher-level applications far simpler as developers are not required to handle framing of their data, but sacrifices the generality and power afforded by streams. Additionally, this design is simpler to implement due to the underlying datagram transport provided by WebRTC and Chord messages.

Choosing a route through the network is made extremely simple by the key-based-routing capabilities of Chord. To generate each (non-terminal) hop of a route, a user needs only to generate a random ID within the key-space. They then determine which node is responsible for this ID, look up its public key within the file system and generate a symmetric public key which will be communicated to this hop. Shallot verifies that each public key obtained hashes to the responsible ID.

Once a route to the destination has been obtained, the client then begins building a circuit across the generated route. To do this, Shallot generates a random circuit ID for the first hop, and creates a build packet directed at the ith hop containing the symmetric key attached to that hop – encrypted using the public key of that node. If $i = 0$, then the client *augments* this build packet with the circuit ID as described below in **Algorithm 3**. It is important to note that the source node knows only *one* of the many circuit IDs along a route, as these are negotiated by the relay nodes themselves. For all other values of i, the client appends a destination field to the build packet so that the $i - 1$th node knows the next destination when it becomes responsible for *augmenting* the packet. The client then wraps the build packet with all propagated symmetric keys as in **Algorithm 2**, keeping this data secure from all but the $i - 1$th and ith nodes. The final node is sent a finalisation packet containing the source node's ID once the route has been extended up to the end – as before, this is onion wrapped to ensure that only the final node can determine its role in the path. Shallot performs each phase sequentially, and once every phase of route propagation completes it returns a usable `Session` object to the application.

Data: Symmetric Keys k, First Hop f, First Circuit ID c, Data d, Index (Current Route Length) i
Result: An onion wrapped message d for delivery over f, protected circuit ID s

```
// Generate an initialisation vector for encryption - this will be
   carried alongside c to obfuscate the circuit ID over each hop.
iv ← randomBytes();
if i = nil then
    i ← k.length − 1;
end
// Now, encrypt for each hop in route from last to first, mirroring
   unwrap order.
while i ≥ 0 do
    d ← symmetricEncrypt(d, k[i], iv);
    i ← i − 1;
end
// Protect the circuit ID and encryption IV to be read by the first
   hop - the latter must be tunneled across the whole route.
s ← f.pubKeyEncrypt(c + iv);
```

Algorithm 2: *Algorithm for onion wrapping a message.*

Although most of **Algorithm 2** does not bear explanation as it is identical to the procedure followed by Tor, the decision to multiplex the circuit ID and initialisation vector together may seem alien. We have two goals to consider: not only do we need to communicate the next circuit ID so that the next hop knows how to handle the packet, we also need to hide this sensitive data as, unlike in Tor, we do not have a service like TLS over Chord. However, simply encrypting this data is insufficient – while it will not be decryptable or directly readable to another node, attackers *will* be able to see patterns in traffic flows as the secured circuits *will appear identical each time without mutation by some salt*. As it happens, this doubles as an efficient way to communicate a different initialisation vector for every onion-routed message.

Shallot makes use of 2 classes of packet between nodes, *relay* and *build* packets (these are analogous to Tor's *create* and *relay cells*). Strictly speaking, these aren't specialised packets – and are in fact calls made through `RemoteCallable`'s RPC framework to cheaply introduce reliability and error detection. This design choice also prevents the owner of a link from having to periodically send "pings" along an active onion route to check for liveness.

On receipt of a *relay* packet, Shallot decrypts the circuit ID and IV from the end of the packet using its private key from Chord, and then uses the circuit ID to determine which symmetric key it will use to decrypt the message body. After decrypting with the correct symmetric key and IV, the module then determines the format of the contained packet: either another relay packet, an incomplete build packet, a finishing packet or a content packet. Additional *relay packets* are sent to the next known hop, with the outbound circuit ID along

the chain being replaced and secured as in **Algorithm 2**. An *incomplete build packet* informs the receiving node what the identity of the next node along the current circuit is – this build packet is then *augmented* as in **Algorithm 3** with a new random outbound circuit ID, before being sent to the new destination. A *finalisation* packet informs a Shallot client that it is the exit node for a route, while reveling to it the source node's identity – alowing a return link to be opened if desired. The exit node then fires the `"receiveConnection"` event with a `new RecvSession(...)`, and connects this to the current circuit. Finally, *content* packets are passed as data to the `RecvSession` attached, firing an event for listening applications.

Data: Own ID id, Next Hop f, Next Circuit ID c, Message m, Secured Symmetric Key $m.k$
Result: A complete build packet m

```
// Remove the field specifying the destination, f.
m.dest ← nil;
// Place the next circuit ID into the packet so that only f can read
   it.
// Include our node ID so that f can verify this packet's origin.
m.circ ← f.pubKeyEncrypt(c + id);
// Sign this packet to prove:
// a) we received this packet from further up the path
// b) the key has not been modified outside of the route
// c) the new circuit is part of this path, and was chosen by us.
m.verif ← id.privKeySign(m.k + m.circ);
```

Algorithm 3: *Process for augmenting a build packet.*

Completed build packets require signing by the last node in the current path as shown in **Algorithm 3** because we cannot leak any information about the source node. This guarantee is sufficient – any alterations to the intended symmetric key will lead to the source node receiving an error notification and being unable to complete the route, closing it instantly. On receipt of this class of packet, a node uses its private key to decrypt $m.k$ and $m.circ$ from the received message, obtaining the inbound circuit ID as well as the attached symmetric key and knowledge of the identity of the last hop. With the last hop's ID, the recipient node is able to obtain the last hop's public key and verify that the message content has not been modified on the Chord network – if the signature is invalid, then the new circuit entry is discarded.

Key lookup through the file system gives us a very strong basis for verifiability provided, of course, that the file system is truly secure. In the event that records on the file system *are* compromised, then Chord's self-certifying identifiers allow Shallot to detect that any tampering has occurred and inform the application that obtaining a route has failed – how best to handle this is left to the discretion of application developers.

Crucially, modules may send all, part or none of their traffic across onion routes – this allows developers to make specific trade-offs between performance and security guarantees within their designs. A developer *shouldn't be forced to* onion-route traffic which doesn't need it. As a result, access to the standard Chord message and module system is not hidden in any way; allowing developers to potentially gain performance by making use of onion-routing selectively.

Presently, a connection is said to be closed if a message transmission fails even once – although this is extremely primitive, this does allow detection of failure to be performed reliably. One caveat of this approach is that the receiving side of an onion link is unable to determine the liveness of a connection.

4.5 Message Client Backend

Building upon all of the services defined in Chord and Shallot, we now have a feature set that will enable the design of a messaging client that fits most of the requirements introduced in **Chapter 3**. The proposed design is based heavily upon *Secure Mailboxes*, with strong reliance upon the distributed file system provided by Chord.

Secure Mailboxes are a concept introduced by Liu et al. in "Tor Instead of IP" [13] – in their design, clients put and get messages from secure mailboxes through onion links, placing a buffer (and layer of indirection) between most forms of direct communication. These can be modified to follow the publisher/subscriber model within the above system – any node which wishes to post a message to a mailbox opens a unidirectional onion link and sends the data across. The mailbox then, in turn, maintains unidirectional links to all nodes which have subscribed to it and forwards this message down each onion link. In fact, the service provided by mailboxes provides a very suitable conceptual basis for communication – both 1-to-1 and in groups.

One of the key requirements for users is the ability to attach a temporary, human readable "name tag" to their cryptographic identity within the chord network. To do this, users place publicly available files onto the network linking their ID to their current name and vice versa, enabling lookup on both keys. Crucially, the node must *own* each of these file records – no other node can be allowed to modify or alter these files, otherwise it would be trivial to impersonate a named user. The same design can be used to connect groups with the mailboxes which they reside upon.

To facilitate fair distribution of mailbox selection, every node in the system must host at least one mailbox, free for any other node to make use of. All messages sent to mailboxes, consequently, must be encrypted by a symmetric key derived from the group name and password, with the group name included in cleartext as part of each message to aid the client in filtering messages and selection of keys for decryption. 1-to-1 communications are performed using the mailbox held by either participant, with an ephemeral password agreed out-of-band. The design additionally allows for mailboxes to subscribe to one another, allowing users to hide behind extra layers of indirection if they feel the onion routing scheme is not sufficient on its own.

It is, however, unclear how to implement global advertisement of users who desire conversation – this runs contrary to some elements of the design, such as file system authentication. Perhaps a suitable scheme would have the global user list be held and maintained by one node through some leader election protocol (such as the bully algorithm) – this may prove incompatible with the defined threat model, however.

4.6 Message Client Frontend

The frontend for the message client would be a simple HTML and JavaScript user interface, designed much in the style of IRC. As the design of this layer is relatively unimportant (in that it does not directly impose constraints on the feature set of lower layers) and this stage of the implementation was never reached, no design specifics are presented here due to time constraints and lack of necessity.

4.7 Summary

Over this chapter, I've shown the complete architecture and design for the network stack, the purpose of each module in the system as well as how the final design meets most of the requirements defined in **Chapter 3**. Additionally, I've shown how my designs have evolved over time to handle challenges revealed by the implementation and imposed by the environment; such as WebRTC's connectivity constraints and edge cases within the various signalling channels. Certain changes, such as the message and module backend, optimisations for Chord and the

state-machine formalisation have been shown to help create a more reliable and extensible system for usage in higher layers of the stack.

The design of a hybrid onion routing scheme between Tor and I2P was detailed – elaborating on the reasons behind such design decisions as unidirectional tunnels combined with circuit-switched route generation. Finally, the core concepts behind a potential implementation for the messaging client were shown. With reference to the network stack features they relied upon, feasible ways to meet many of the design requirements were described.

Chapter 5

Implementation Details and Limitations

Working from the design introduced in **Chapter 4**, this chapter focuses on many of the points encountered while implementing the network stack such as any difficulties, the choice of particular cryptographic algorithms, limitations relative to the design and the limitations of the design itself. Additionally, this chapter will discuss my testing methodology for the project. In most cases, potential methods to improve current designs and to implement and provide missing functionality are suggested in **Section 8.2**.

5.1 Difficulties Encountered

JavaScript, although mandated by the use of WebRTC and the choice of deployment environment, lacks many of the features that would have helped in the development of a large-scale distributed system such as language-level remote procedure calls. Additionally, in JavaScript all numbers are stored in double-precision floating point representation, giving a maximum safe integer size of $2^{53} - 1$. By contrast, Chord requires the usage and manipulation of arbitrarily large (n-bit) integers. This required me to develop my own implementations of these crucial features.

The decision to work with a unified codebase for both client and server deployment (the browser and Node.js respectively) presented some technical challenges – although it did make the design and implementation easier overall. In particular, Node.js has no native support for the WebRTC protocol or API: any implementations of this are provided as open-source C++ addons by the community. As a result, my designs had to be adapted to handle multiple "adapters" for this functionality. Two of these such adapters were used within the project – one to unify the API differences between browsers, and one to introduce WebRTC functionality to Node.js. While the former was mostly perfect for my uses, the latter had fallen behind the W3C specification for the WebRTC API [8]. As a result, I had to spend a portion of time within the project on modernising this package's API.

Integration of multiple cryptography libraries throughout the project was a particular source of trouble. Although these modules perform ostensibly similar tasks and operations upon data, the input and output formats between any pair of modules varied so wildly that reconciling these differences took large amounts of trial and error alongside inspection of their respective code bases. The use of several different libraries *was* necessitated by my particular choices of hashing and cryptographic algorithms, but had I known how much trouble this would introduce I'd have altered my design sooner.

5.2 Conductor

Presently, Conductor does not handle efficient deallocation of connections through reference counting or any similar management scheme. As a result, handling of connection lifetimes is completely left to higher-level applications, making its use considerably less convenient. Conductor detects disconnection by watching for any of the 3 (recoverable) failure states in WebRTC, "closed", "failed" and "disconnected". A more mature system would likely include some form of handling for the concepts of temporary and permanent disconnection – potentially by making use of some configurable timeout period to demarcate the two states.

Additionally, Conductor is presently unable to detect disconnection events in this fashion in Mozilla Firefox – this is a bug within the browser itself, which has been present since 2013 [18]. For other browsers, neither Safari nor Internet Explorer (Edge) are compatible with the protocol – this heavily impacts the browser cross-compatibility of any dependent services.

5.3 Chord

Implementation of Chord required much deliberation on the choice of cryptographic algorithms within the scheme with regards to modern security findings. While the Chord paper makes (and recommends) use of SHA-1 for hashing with a 160-bit identity space [22], it has come to light in recent years that the algorithm is fundamentally insecure. In particular, an attack by Stevens, Karpman, and Peyrin presented a significant breakthrough for cryptanalysis of the algorithm – by running attacks on a relatively cheap, high price/performance ratio Graphics Processing Unit (GPU) cluster, they estimate that a full SHA-1 collision could be obtained for between 75K$ and 120K$ worth of computation time in late 2015 [21]. Not only is this below the cost bound they claim was expected to be achieved *by 2018*, this is deemed to be well within the resources of a nation-state or criminal syndicate. Within my design, if a collision can be found against the identity scheme then it may be possible to fool the various verification systems in play. Choice of a hashing algorithm for use in Chord must also balance the width of the identity space against the security of the chosen hashing algorithm – an m-bit space requires a finger table of size m, when running many connections could impose a high resource and bandwidth cost on browsers. For these reasons, ***SHA3-224*** is chosen as the base hashing algorithm within the system – no known attacks exist upon it, the design aims to maintain collision resistance, and the key-space is not significantly wider than that recommended by Stoica et al.

The asymmetric cryptographic algorithm chosen as the basis for Chord's identity scheme is ***RSA-1024*** for multiple reasons. The algorithm is well-understood, and it relies on computational problems which are deemed sufficiently hard to thwart direct cryptanalysis. The smallest available key size is chosen due to the performance cost that RSA imposes, both to generate a new key-pair at initialisation and for encryption and decryption of data. In practice, key-generation *does* impose a very high, variable time execution cost at initialisation within the browser, likely due to the JavaScript environment. Throughout Chord, this is used in combination with ***AES-GCM-128***, a symmetric encryption algorithm, where faster data encryption is needed. This algorithm is far faster and is similarly proven throughout its long history, relying on RSA to encrypt the shared symmetric key – this is used primarily within the Conductor Channel implementations, in a manner similar to TLS.

In practice, the adapter for WebRTC within Node.js has proven temperamental at best. While JavaScript is designed to be perfectly safe and sandboxed, this open-source extension has been notoriously buggy at the most inopportune times. Segmentation faults arise for no reason (despite the fact that this should be an impossibility within a safe language like JavaScript), crashing the Chord server at unpredictable and inappropriate times. For instance, refreshing a webpage while connected to the server (a normal action as far as users are concerned) will crash the connected server with a 100% rate of reproduction. As a result, the system is nowhere near as stable as a finished product should be – I deemed it far beyond the scope of this project to repair the internal C++ code of

an open-source project.

At present, the message- and module-based delegation system expects that all nodes within a network run the same set of services and modules. The design of all modules presently makes the assumption that any message can be served and handled by the directed recipient – a more heterogeneous system would require some additional work, but I believe it would be possible to adopt such a paradigm.

The proxying functionality within the message core currently assumes that a message will be proxied at most once along its path to a given destination. In particularly odd cases associated with high churn, it is likely that a message could require multiple proxies to reach the core ring; in which case the receiving node will need to know the complete proxy list to reply reliably. A valid approach might be to modify the packet format to account for multiple proxies (up to a sensible limit). The messaging rules also make the assumption that proxying over any node is guaranteed to eventually direct the message to the chord ring – this guarantee is mostly sensible, except in the "External" states where it may make more sense to directly ask any connected node about its current state and choose the "most connected" partner as a proxy.

In the event that a node fails to create a connection to another node, then my present implementation of Chord does not account for this and detect failure. The functionality to handle this *is* programmed into Conductor – the trouble arises when trying to set a sensible upper bound for a timeout and connection establishment time as the network grows. Similarly, Chord does not actively close any connections which are not needed due to Conductor's limitations, potentially making its operation very expensive at scale.

Within the file system, the implemented process for securing sequence numbers differs slightly from the initial design. Due to difficulties surrounding representation of numbers within JavaScript, the sequence number and circuit ID are encrypted in JSON format for simplicity. Additionally, files are not replicated across the file system to provide reliability, due in part to time constraints, and due (more significantly) to security concerns introduced by file ownership and authentication. Implementing this key feature would make the system more reliable, but would likely require significant design work. As a direct consequence of this absence, peers take all files they are responsible for offline as they disconnect – there is no handling for exits from the Chord ring, both controlled and uncontrolled. Additionally, files have no protection from the node which is responsible for storing them: in the current system, an attacker could modify any file which they are tasked with storing. It is possible that files could be signed by their authors as a countermeasure, at the cost of making file ownership public. To defend against both of these possibilities, Chord periodically checks the file system to see whether its key is still accessible and has not been modified.

5.4 Shallot

In contrast with other sub-systems of Chord which use encryption, Shallot makes use of ***AES-CBC-128*** rather then AES-GCM. Although GCM is typically considered to be a safer mode as it generates additional tags for data verification, it becomes very expensive to have to bundle many tags as part of each encrypted datagram. By using CBC instead, only the value of the random initialisation vector shared among all encryption layers must be transmitted alongside the message.

There are numerous limitations to my onion routing implementation. Link failure is only detected once a message delivery has failed, and in effect the receiving side of a session is unable to detect this link failure. Packets are not constant size and send rate is not constant – following from this, the module's onion traffic is extremely vulnerable to traffic analysis. Realistic onion routing systems *do* provide this functionality, and it is proven to be very effective at thwarting traffic analysis where passive monitoring is concerned – this would be extremely worthwhile to implement in future.

Another limitation of Shallot at present is that in-order delivery of messages is not guaranteed, although

by the current link failure constraint, delivery itself *is* reliable, acting like a hybrid between UDP and TCP. I don't feel that this needs to be handled at Shallot's level – applications which desire such semantics are free to implement it themselves.

The current onion routing algorithm over-privileges the forward direction when generating new circuits – an attacker forwarding to the same node multiple times may cause a link to fail by choosing the same output circuit number twice. This allows for a potential Denial of Service attack – further implications are explored in **Chapter 6**. It may be worthwhile to adapt the process to get confirmation from the next hop before using a circuit ID, or alternately derive the circuit ID through some handshake.

5.5 Testing Methodology

Integration and unit testing of Conductor and Chord was performed through a MochaJS test suite, automated by TravisCI in response to any commits to the GitHub repositories of each module. Although testing of WebRTC related features could not be performed, connection and channel management logic for Conductor was tested in its place. Unit testing of Chord was enabled without the use of WebRTC for the most part by testing all operations on a 1-node ring, which allowed testing of the file system. Automated testing of the implemented Identity library for n-bit arithmetic was also performed in a very thorough way, having no reliance upon network functionality and being well-suited to testing. Shallot has no unit or automated tests at present.

Testing of the complete system was performed in a mostly ad-hoc fashion. Due to my lack of access to suitable infrastructure to conduct large scale testing for validity and performance, all of my testing occurred between two machines: my home computer (Windows 10 x64, Google Chrome/Firefox, Intel i7-920 @ 2.80GHz [4 Core], 6GB RAM) and a remote Digital Ocean VM (Ubuntu 14.04 x64, Node.js, Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz [1 Core], 512MB RAM) hosting all clients and the server respectively. Locally, I hosted a maximum of 15 clients in separate browser windows. This is because when multiple tabs within the same window were used, Google Chrome was observed throttling unseen tabs, causing message delivery delays and timeouts on periodic actions to fire. File system accesses and onion routing both appeared to run acceptably in this set-up, with very little (humanly perceptible) delay noted. Unfortunately, no hard performance numbers are available.

I had intended to make use of all machines I have access rights for within the School of Computing Science to generate a large-scale testbed for the project. As it turned out, the WebRTC adapter for Node.js could not compile on the school machines as it required a higher version of glibc than was available. I conveyed this to my supervisor, who confirmed that it would not be possible to upgrade the version in use within the School due to previous issues introduced by the last attempted large-scale system upgrade.

5.6 Summary

Although JavaScript's language deficiencies presented trouble during implementation, they could be bypassed without too much work. Implementing these features was informative and useful. Importantly, many of the limitations discussed above affect not only scalability, reliability or performance, but also lead into practical attacks on the system – these vulnerabilities are elaborated upon in **Chapter 6**. However, many of these limitations merely affect cross compatibility between browsers – and in turn, the measure of how deployable a WebRTC solution is in practice. Although it is hoped that browsers will fully implement the spec in due time, this is not an ideal situation.

Ad-hoc testing was sufficient for showing basic operation and for development of the system – but ideally,

more rigorous quantitative testing is required. The main obstacle to this goal is suitable infrastructure; real world testing needs to be done at scale above all else.

Chapter 6

Security Considerations

This chapter provides a more critical examination of the security guarantees and weaknesses of my network design, discussing how several aspects of the design add to the system's security. Crucially, several practical attacks are presented against the implemented design, displaying the known weaknesses of the system alongside potential modifications to remedy them.

It must be stated beforehand that this analysis is by no means exhaustive – I am not a security professional, and there are almost certainly attacks or other uncertainties I have missed or which go beyond my understanding of attack construction.

6.1 General Considerations

The use of self-certifying identifiers within the system greatly enhances the security guarantees that can be made. Assuming proper verification is performed, identity spoofing is made impossible due to the massive 1024-bit key size provided by RSA. The choice of an unbroken cryptographic hashing algorithm such as SHA3-224 ensures that finding a public key which collides with a known node's ID is computationally intractable on modern hardware with presently known and disclosed techniques.

I'm unable to make any observations or guarantees about the security implications of the design of the messaging client. Since it has yet to be implemented, it's very difficult to make any claims when the design has not yet been tempered by real-world considerations.

6.2 Design Weaknesses and Known Attacks

Although the use of self-certifying identifiers is useful in practice – we gain source verifiability, and generation of additional RSA public-private key pairs is by no means free for attackers to perform – attackers remain able to mount a *Sybil attack* on the system. A *Sybil attack* is defined as an attack where safeguards and reputation systems are bypassed through the generation of a large volume of identities – attackers may pre-generate and cache vast numbers of valid identities until they own a set which is evenly distributed around the identity space (*full-space*) or until they obtain an identity which belongs to a desired region of key-space (*partial-space*). This can allow an attacker to take responsibility of a file if they so wish, or to arbitrarily increase the likelihood that they are chosen as part of an onion route; Sybil attacks are assumed to be the precursor for many of the attacks on the design which follow. Weakness to Sybil attacks is not exclusive to my design or implementation, attacks

have been seen in the wild against Tor [2] and some level of weakness is theorised in I2P [27]. Approaches such as the Invisible IRC Project (IIP, not to be confused with I2P) use proof of work schemes to limit the amount of identities that can be generated as a defence against Sybil attacks [26]. Most importantly, Sybil attacks become harder to coordinate and execute as the network size grows [27] – making a small-scale deployment such as Shallot particularly vulnerable.

6.2.1 File System

When storing a file, a request is susceptible to having the included public key substituted for that of another node. An attacker may use this and substitute their own public key into the packet, either to take exclusive ownership of a file or to co-opt a file's ownership by secretly storing the returned ownership key before forwarding it to the true owner. The design of this action requires adaptation to either send this information securely through signatures with additional layers of encryption on call parameters or to lookup the owner's key through the central file system and the key-management module suggested in **Section 8.2**.

In addition to this, file storage and update requests in the network use no form of data signature to protect the integrity of new versions of files. As a result, any node along the path which must handle or direct such a request may freely substitute the file contents with whatever it sees fit. For updates in particular, this can be remedied by including and hiding the hash for the *new* data's contents rather than the old inside of the $proof$ for the intended recipient. For new files, the request must include some form of signature to ensure that ownership cannot be co-opted, but how to prevent *its* modification across the network in turn is unknown at this time.

Ownership and storage transference through key relocation are NOT absolute – in the current design, the prior storage location or owner may still retain the ownership access key. It may be worthwhile to examine the approach of [3] to gain some insight on how to efficiently perform key revocation to defend against ownership transfer between two nodes. Additionally, it may be possible for file ownership to be managed with asymmetric keys – where the file owner is in possession of a private key with which they can use sign updates, verifiable by the store which holds the public key. In practice, such a scheme might be too expensive to enact due to the variable length delays I've experienced while generating identities for nodes.

Files in the present scheme are modifiable without repercussion by the node responsible for their storage. Although they do own a copy of the ownership key, naturally they have no reason to use or check against it should they wish to modify a file's contents. This could be fixed by having files bundled with signatures created by their owner, allowing any node downloading a file to detect whether or not tampering has been committed by the responsible node. Similarly, an attacker can take responsibility of an existing file on the network and refuse service of it via a partial-space Sybil attack. Assuming they can no longer inappropriately modify the file, this allows an attacker to feign ignorance about a file's existence, denying access to it as part of a Denial of Service attack. This could be fixed with file replication across the responsible node's successors, but remains vulnerable to sufficiently advanced Sybil attacks which occupy all the available successors for a file. To defend against this, a suitably large degree of replication is needed – as a result, it becomes harder for an attacker to generate enough keys in the right regions of identity-space to an arbitrary level of precision to deny all service.

6.2.2 Shallot

As discussed in **Section 5.4**, the current route building algorithm over-privileges the forward direction when specifying circuit IDs. By choosing the same circuit ID twice, an attacker forwarding to the same node multiple times may cause a link to fail when the next hop is unable to decode the message. Although an attacker cannot selectively use this attack with any fine precision, this may be useful as a blanket Denial of Service strategy to knock out all onion-routed communication. As a fix to this, it may be worthwhile to get confirmation from the

next hop before using a circuit, or by generating circuit IDs through negotiation or consensus. This attack *is* easily detectable by the source node, who will immediately realise that the link is closed once message transmission fails.

Attackers can exploit the remote procedure call framework's acknowledgement semantics to create a silent Denial of Service attack on any onion routes which pass over them. To do this, an attacker is expected to begin with a full-space Sybil attack to increase the likelihood that they are chosen as at least one of the relays along a route. From here, an attacker may respond to any "relay" call made upon it with any answer message, fooling all higher levels of delivery into believing that the message transmission was successful. As a result, the source node never discovers that traffic is being lost and never closes the link in response. Again, an attacker is unable to use this technique selectively in most cases, but it presents a more insidious potential attack than the one discussed above. it may be possible to alleviate this with the aid of some challenge and acknowledgement that only the final node is able to respond to correctly – in the event that an incorrect acknowledgement is received, the source endpoint becomes aware that such an attack is in progress. Additionally, periodic regeneration and rotation of onion routes might prove to be a viable strategy, assuming there are enough users to minimise the chance of selecting a "bad" node.

Within the system presently, nodes do not send packets at a constant rate through techniques such as noise generation or packet buffering. As a result, there is very little extraneous traffic which can be used to mask the routes chosen for onion relaying, potentially presenting a vector for an attacker to conduct traffic analysis. If an attacker is able to monitor and determine traffic flows (i.e. with a sufficiently large set of controlled clients through a full-space Sybil attack) then they may be able to determine the source and destination endpoints, to prove that communication is taking place.

The above Denial of Service attacks can be targeted more selectively, but with far greater difficulty by analysing packet sizes. Presently, packets grow in size with successive layers of encryption due to padding both pre- and post-encryption. As packets are not divided and sent as uniform chunks within the system, packet size can be used as a probabilistic indicator of distance from either endpoint – large packets are more likely to be observed closer to a source, and smaller packets are more likely to be seen at relays closer to the destination. Using this observation, an attacker who is made part of an onion route could, over time, estimate their certainty that the last or next hop along a route is the source or destination endpoint and conduct any of the above attacks. The most obvious counter-measure would be to adopt a single-packet-size-model as employed within Tor.

Crucially, the model does not allow attackers to ever read the contents of any onion packet. As end-points of any communication *are the destinations themselves*, unlike with Tor exit-nodes, the contents of any packet are never exposed to external scrutiny as they leave the network.

6.3 Best Practices

When working with the remote procedure call framework, developers should expose only a subset of a module's functionality remotely if they wish to reduce the attack surface exposed to adversaries. Validation of input and prevention of unintended actions is time-consuming and very difficult to ensure for each and every remotely available action – by reducing a module's public API to fewer operations, developers are able to dedicate more of their time to the management of fewer safety checks, while reducing the amount of potential attack vectors exposed to adversaries. Ideally, any application developer should either make remotely available procedures idempotent and functional in nature, or they should make a strong effort to minimise any unverified mutation of state where possible. Verification of answer content may also be necessitated to prevent tampering of tunnel-based transports built with the framework.

6.4 Summary

Broadly, the current system design provides very strong protection against impersonation and identity spoofing through the usage of self-certifying identifiers. However, all the presented attacks on the file system invalidate the guarantees we need for the system design to be sound – applying the proposed fixes and design modifications in future is crucial for the system to be secure.

By directly drawing influence from established protocols, the only attacks on the implementation of onion routing itself are Denial of Service attacks. Although they make usage more difficult and preventable for users, the underlying data integrity is never compromised. Although potential attacks are theorised to allow partial unmasking, it remains unclear whether or not these Sybil-based attacks are feasible in practice.

With regards to the onion-routing implementation provided by Shallot, it is unclear whether or not any of its differences from Tor, I2P or other such designs introduce additional unseen vulnerabilities or compromise the guarantee of perfect forward secrecy.

Chapter 7

Evaluation

This chapter summarises my experiences throughout the project in constructing my original design, particularly concerning the design of secure systems. I also talk briefly about my thoughts on working with the intended design, and discuss the knowledge and experience I've gained throughout the project.

7.1 Project Design

Chord, against my best wishes, represents the most significant time investment in the project. While it made higher levels easier to develop and program securely, it led to constant difficulty during its development; not only because of the deviation from the Chord specification but due to the specifics of WebRTC and the browser environment. Although I'd prefer to have composed my design without it, Chord proved to be an extremely powerful architectural basis, while also presenting unique challenges that have taught me a lot about the design of distributed systems. Its importance to the design cannot be stated enough, Chord provided many opportunities for adaptation of simple functionality such as distributed file storage into versions adapted with security in mind.

In hindsight, trying to target both Node.js (server) and the web-browser (client) environments on a single code base was an awful idea, but an admirable one nevertheless. Although I saved myself the trouble of writing and maintaining two different-language projects with identical feature sets, the project ended up being extremely reliant on external open source projects. Broken in subtle and often insidious ways more often than not, I had introduced sources of error into my program that were effectively out of my control due to my inexperience with C++. Further valuable time had to be dedicated to modernising the API of one of these libraries to match the W3C specification due to the unstable nature of WebRTC.

7.2 Choice of Implementation Language

Because of the project's primary focus and goals, JavaScript was the only suitable language for implementation (in the client, at the very least). Despite JS being one of my favourite languages for game development (including online game development) and prototype development, I've come to the conclusion that the language is an abysmal fit for the design of large-scale distributed systems. The distinct lack of language support for remote procedure calls as well as large number (n-bit) arithmetic meant that I had to develop these features from scratch within the language. Although I gained a lot of valuable experience from this "do-it-yourself" approach, I have to acknowledge that I'd have had more time to improve on the rest of the system, to think about the design and to address glaring flaws throughout the project.

Although I'd liken the use of JavaScript in the development of complex network topologies to building the Vatican from matchsticks and glue, recent evolutions to the language made its usage far simpler and more suited to the task. ECMAScript 6 (ES6) is the next version of the JavaScript language, and has seen increasing implementation since its proposal. One of the key features introduced is that of "Promises", which are a more sensible and composable abstraction around asynchronous actions than the traditional JS concept of callbacks. Promises lie at the heart of the design of almost all elements of my design's implementation for this reason – the amount of mental work they alleviate is immeasurable. Additionally, they allow for far richer remote failure modes. The addition of classes and inheritance to the language by ES6 make structured programming within the language far simpler while also allowing advanced concepts like dynamic trait mixins – although the world at large is slowly moving away from the Object-Oriented paradigm, in certain classes of application these language features still aid greatly.

JavaScript is traditionally immune to race conditions by following a single-threaded event-driven execution model. When considering a multi-user distributed system where each runs periodic events without consideration for synchronisation between nodes, however, these guarantees fall away. Debugging and systems design in such an environment with cross-platform considerations becomes very hard, very quickly. A lot of my debugging work for the message routing rules involved deduction and pen-and-paper work more than interaction with a debugger due to Node.js's abysmal debugging tools. When working with modern language features and asynchronous calls, the browser's JavaScript debugger doesn't often fare much better – though it is far better than having nothing at all.

7.3 Experience Gained

Ultimately, working on this project has given me a lot of valuable insight into the design and development of security-focused protocols and applications. In particular, analysing my own designs for flaws and attacks from an adversarial perspective has shown me the difficulty of creating a trustworthy, reliable design for any part of a large-scale system. Implementing features like RPC from scratch, while time consuming, has helped me to develop an understanding of how such systems work in practice.

Development of effective routing rules took some amount of thought, as did the formalisation of Chord with very little help from the specification. The Chord paper is simply not written with a WebRTC-like model in mind, meaning that the design of a suitable state machine had to be concocted with great care, consideration and reliance upon my intuition. As a result, I'm far more likely to make use of state machines and other formalisation methods in my future work – assuming the system is correctly modelled, the paradigm is extraordinarily powerful for building reliable systems.

7.4 Summary

Overall, working on this project has given me a lot of insight that I will carry forward with me into my future work. Although I feel that certain aspects of the design were not the best choices in hindsight, collectively they *do* provide a coherent and powerful system. Although I have my grievances with using JavaScript, the language mandated by WebRTC, I still believe that it is a powerful technology for building smaller-scale systems.

Chapter 8

Discussion and Future Work

This chapter concludes the report, connecting the original project goals with my findings, experience and results. To add to this, enhancements and future developments to the design and final implemented product are discussed at length.

8.1 Conclusion

Throughout this report, I have ultimately shown that onion-routing over WebRTC *is* a feasible approach, and that the browser is available as a suitable environment for deployment. However, my experience throughout the year has shown me that the tooling is perhaps too immature for this to be immediately realisable for widespread or commercial usage – parity of WebRTC feature support between browsers is lacking, the lack of proper WebRTC support for serverside JavaScript environments and JavaScript's current unsuitability for large-scale distributed systems hampered this project on many occasions. Additionally, my design is rather immature and contains many glaring security holes, which I've explained where possible.

While the final results of the project are defined more by their limitations than anything else, the work I've presented here demonstrates a *proof-of-concept* for bringing the onion routing paradigm to the browser while showing the power of WebRTC. I hope that this encourages others to examine different architectures and designs. Although I believe that implementing many of my proposed fixes to the design shown here will eventually lead to a secure basis for a system, it is more than likely that a design based on a completely different architecture would encounter far simpler or more elegant solutions to the problems I've encountered.

I believe that modifying Chord to make use of a message- and module-based approach is a powerful adaptation of the protocol. Specifically, it accounts for the design of more traditional classes of application that wish to leverage its peer-to-peer topology in novel ways. Although the additional enhancements presented (such as the authenticated file system) have many present vulnerabilities, additional work to repair the design could add to the protocol's usefulness.

8.2 Future Work

First and foremost, the proposed messaging client is to be implemented. This will allow me to assess and discuss the viability of the core application design as well as fulfilling one of the original project goals of making secure communication over onion-routing deployable. A collection of libraries alone is by no means usable for

most clients. Additionally, this will demonstrate whether or not mixing onion-routed communication with more traditional paradigms within a single applications holds merit.

Conductor can be enhanced beyond its current capabilities. Better modelling for WebRTC's more nuanced disconnection model may prove useful in practice, by making networks built on top of it more reliable. More efficient resource management could be obtained by combining reference counting on each end alongside a dedicated backing channel. Conductor would periodically exchange information between each end of a connection to determine whether it remained in use at either side of the link. Attaching an operation to the browser's page exit handler would also allow for instant disconnection detection in the event that the browser or tab is closed by the user.

My adaptations to the Chord protocol will need considerable work before they are truly suitable for secure systems design (**Chapter 6**), but there are further enhancements that can be introduced to make the system more reliable, fast and scalable. It may be worthwhile to move all key lookup and handling to a central module of Chord – Shallot already makes use of such functionality internally to aid lookup of known keys, for instance. This could be combined with the additional traffic classification – separating messages which require security and verifiability through mandatory signatures from the source node, from those that do not such as in self-certifying IDs. The system use these different classes of traffic intelligently; some remote calls such as `.notify()` within Chord would strongly benefit from source verifiability, but to enforce these guarantees blindly could have an impact on performance due to the necessary key lookups. With the built in support for multiple packet formats within the system, moving to a binary message format within Chord rather than the current JSON would be of great interest. This would reduce packet sizes, making message transmission faster and less bandwidth-intensive throughout the system due to JSON's syntax overhead.

Additional work would need to be undertaken to improve fault tolerance across the network in line with the recommendations of the initial specification [22]. A key concept to implement would be file replication – distribution of multiple copies of a file throughout the network in the event that a responsible node disconnects and a file is lost. However, it is unclear how this would interact with file ownership and authentication of updates – further design work would be mandated to make this a reality. Additionally, nodes could maintain backup lists of successors to aid rediscovery in the face of high churn – my state machine does model for this, but the functionality could not be included due to time constraints. Further enhancements are possible from this, such as periodically contacting known server nodes across Chord to detect network partitioning or pinging connected nodes to check for liveness.

The current iteration of Chord assumes that every node runs an identical set of modules, with identical service support on every client and server node. Adaptation of my system to include heterogeneity of services and capabilities throughout the system would have far reaching consequences, enabling a single multi-purpose network for many different applications, potentially bolstering the user-base and providing more protection from Sybil attacks. This would also allow the server to have a more light-weight design than all other nodes within the system, enabling its development in a more robust language such as C++ – allowing my design to escape its current reliance on broken open-source adapters for WebRTC in Node.js

To combat the security vulnerabilities discussed in **Chapter 6**, multiple approaches are considered in addition to those already suggested earlier. Introducing the measure of a node's "trustworthiness" to the system might prove to be an effective way of handling rogue nodes throughout the ring – nodes which repeatedly commit protocol violations or are involved in a disproportionately high amount of errors may be excluded by their peers through some consensus-based protocol. Ownership within the file system could be migrated to a public-private key system to handle file relocation in a safer manner, although RSA would likely prove too expensive in practice. Accesses to public keys from the file system should be modified to work though the above proposed central key storage module. Signatures and verification would need to be more prevalent throughout the file system to prevent illegal substitutions and modifications of sensitive data.

The onion-routing scheme presented in Shallot needs further modification to be securely and reliably used.

The current weaknesses to traffic analysis are unacceptable with respect to one of the main goals of onion routing itself – the next steps for the development of the module include a move to a constant rate of data transmission, compensating for quiet periods and excess load through random noise transmission and packet buffering respectively. A move to a more traditional approach such as Tor's may be a valuable avenue in future – particularly as it is unclear if any of the minor modifications made to Shallot's protocols affect the guarantees of perfect forward secrecy.

Examination into existing methods of mitigating and impeding Sybil attacks on the system would make the network more resilient against many of the offensive techniques that they enable. Although proof-of-work schemes such as *HashCash* within the *Invisible IRC Project* are suited to this task, their computational overhead may prove prohibitive for deployment within the browser. Additional research in this field is warranted.

Finally, rigorous performance testing of the P2P overlay and onion-routing system will be required to show whether the design proposed throughout this report is usable in practice. In particular, it would be worthwhile to see how latency and maximum bandwidth scales with network size and onion route length. Crucially, these numbers must be compared against those observed in Tor and I2P to provide a more direct comparison of the effects that deployment environment and protocol have on these metrics.

Appendices

Appendix A

Installing, Compiling and Running the Program

Provided within the "L4proj-15-16-SIMPSON,kyle.zip" file, are four subfolders – "Conductor", "Chord", "Shallot" and "shallot-test". The first of these 3 correspond to the libraries of the network stack I have developed and are designed to be included as part of some higher level application – they are not directly usable. The folder "shallot-test" *DOES* allow interaction and use of the network, and contains server and client programs for your use, if so desired. Instructions on their use are included in the README.md files in each directory.

Compiling the client or running the server will require *Node.js* (`https://nodejs.org/en/`) – this acts as the package management and compilation environment for the client, and as the server runtime. Additionally, the server *must* be run from a modern version of Linux – the school computers are insufficient. A precompiled version of the client program is provided through "compiled.js" and "index.html".

A.1 Testing the Libraries

Both Conductor and Chord have a suite of unit tests to prove operation. To run either test suite, execute the following from inside the module's root directory.

```
> npm install
...
> npm install -g mocha
...
> mocha
```

Test runner output will appear in the console.

A.2 Compiling and Running the Client

To compile the final JS client, from within "shallot-test" open a new terminal window and type:

```
> npm install
...
> npm install -g webpack
...
> webpack uncompiled.js compiled.js
```

To run the client, access "index.html" either directly or by placing it upon a web server in the latest version of Google Chrome – ensure that "compiled.js" is in the same folder as the html page!

The page will then ask for a server address: "ws://mcfelix.me:7171" resolves to my server, which I will attempt to run where possible. If you're running the server locally, type "ws://localhost:7171". **If you are unable to compile the server for your own purposes and my server is down, contact me immediately.**

Press *F12* to bring up the developer tools – select the JavaScript console. The Shallot object exists within the window under the name `s`. To see the current node's client ID, type `s.chord.id.idString`. To open an onion connection, perform the following steps from within the JS console:

```
var otherID = //... the target node's ID.

/* ... */
// After you see "[CHORD]: JOINED!!!!" in console

// Space for the new connection.
var myOnionConn;

// Create a new connection
// Note: whitespace is unimportant here.
// This will work as one line if necessary.
s.connectTo(otherID)
    .then(
       result => {
          myOnionConn = result;
          alert("Connected!");
       },
       error => console.log(error));

// NOTE: The above is how to utilise javascript promises,
// where param => {/*...*/} is a lambda

/* ... */

// Once the connection is created (the above request is async)
myOnionConn.send("Hello world!");
```

The example is designed to hook up handlers automatically, such that another browser will display notification that it has received data in its console. Note that the server has no such code – while you *can* open an onion link to the server, it has no handling activated.

A.3 Installing and Running the Server

In addition to Node.js, the server requires the installation of a WebRTC adapter – "node-webrtc". Compilation of this will take a considerable amount of time as it as a C++ extension for Node.js. The full list of its dependencies is available at https://github.com/js-platform/node-webrtc in the readme file. If you are unable to compile, please do not hesitate to contact me.

To install the dependencies and adapter and to run the program, type the following sequence of commands.

```
> npm install
...
> npm install https://github.com/js-platform/node-webrtc.git#develop
...
> node server.js
```

The server can be observed working when vast amounts of output appear in the console. If you are unable to compile on your environment, again, do not hesitate to contact me!

Appendix B

Chord Packet Format

Messages within the modified Chord scheme have a 2-byte version field, followed by any arbitrary format. So long as it encodes the necessary information required by the system and the receiving node has a valid handler for the delivered packet format, then a packet is deemed valid.

There are two classes of packet in the system design presently – Data and Proxy packets. All packets must (at present) contain type, source, destination, remaining hops and data fields, with an optional proxy field. Additionally, Data packets must have valid module and handler fields, to inform the recipient about how the message must be parsed. Proxy packets, when delivered, are unpacked to obtain a data packet – the recipient then places its own ID into the packet's proxy field before retransmitting.

At present, there is only one packet version: "00", which corresponds to a JavaScript Object Notation (JSON) format for simplicity. Identities are stored in Base64 format, and field names are shortened to one character to save space where possible – these packets are also expanded and formatted for readability. Examples follow, assuming a 224-bit identity space.

KEY

- **Type** - "t"
- **Source** - "s"
- **Destination** - "d"
- **Remaining Hops** - "H"
- **Data** - "D"
- **Module** - "m"
- **Handler** - "h"
- **Proxy** - "p"

Data Packet

```
00{
    "t": 0,
    "s": "F4qaM4j0Gb3zKq23SDEtJqDY0mN1K/0ahfOgyg==",
    "d": "Z7ydoyjca7RNF3QptGcHigBYhZVqGSpHROVjhQ==",
    "H": 245,
    "D": "arbitrary-data",
    "m": "Handling_Module_Name",
    "h": "Delegated_Handler",
    "p": "RNTR/Fmzj38AUNRSP5s7K5hblgc4loqYsUi8nA=="
}
```

Proxy Packet

```
00{
   "t": 1,
   "s": "Z7ydoyjca7RNF3QptGcHigBYhZVqGSpHROVjhQ==",
   "d": "RNTR/Fmzj38AUNRSP5s7K5hblgc4loqYsUi8nA==",
   "H": 123,
   "D": "00{\"t\":0, \"s\":\"...\", ... }"
}
```

Bibliography

[1] Harald T. Alvestrand. *Overview: Real Time Protocols for Browser-based Applications*. Internet-Draft draft-ietf-rtcweb-overview-15. Work in Progress. Internet Engineering Task Force, Jan. 2016. 22 pp. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-overview-15.

[2] arma. *Tor security advisory: "relay early" traffic confirmation attack*. July 30, 2013. URL: https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack.

[3] Agapios Avramidis et al. "Chord-PKI: A Distributed Trust Infrastructure Based on P2P Networks". In: *Comput. Netw.* 56.1 (Jan. 2012), pp. 378–398. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2011.09.015. URL: http://dx.doi.org/10.1016/j.comnet.2011.09.015.

[4] Matthew Caesar et al. "ROFL: Routing on Flat Labels". In: *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '06. Pisa, Italy: ACM, 2006, pp. 363–374. ISBN: 1-59593-308-5. DOI: 10.1145/1159913.1159955. URL: http://doi.acm.org/10.1145/1159913.1159955.

[5] Alissa Cooper. *Real-Time Communication in WEB-browsers*. https://datatracker.ietf.org/wg/rtcweb/charter/. Accessed: 2016-03-19. 2011.

[6] Alan Dearle, Graham N. C. Kirby, and Stuart J. Norcross. "Hosting Byzantine Fault Tolerant Services on a Chord Ring". In: *CoRR* abs/1006.3465 (2010). URL: http://arxiv.org/abs/1006.3465.

[7] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: The Second-generation Onion Router". In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004, pp. 21–21. URL: http://dl.acm.org/citation.cfm?id=1251375.1251396.

[8] W3C WebRTC Working Group and Ian Hickson. *WebRTC 1.0: Real-time Communication Between Browsers*. https://www.w3.org/TR/webrtc/. Accessed: 2016-02-09. 2016.

[9] Christer Holmberg, Goran Eriksson, and Stefan Hakansson. *Web Real-Time Communication Use Cases and Requirements*. IETF RFC 7478. Oct. 2015. DOI: 10.17487/rfc7478. URL: https://rfc-editor.org/rfc/rfc7478.txt.

[10] Randell Jesup, Salvatore Loreto, and Michael Tuexen. *WebRTC Data Channels*. Internet-Draft draft-ietf-rtcweb-data-channel-13. Work in Progress. Internet Engineering Task Force, Oct. 2015. 16 pp. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-13.

[11] M. Frans Kaashoek and David R. Karger. *Koorde: A simple degree-optimal distributed hash table*. 2003.

[12] David Karger et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[13] Vincent Liu et al. "Tor Instead of IP". In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. HotNets-X. Cambridge, Massachusetts: ACM, 2011, 14:1–14:6. ISBN: 978-1-4503-1059-8. DOI: 10.1145/2070562.2070576. URL: http://doi.acm.org/10.1145/2070562.2070576.

[14] Zhiyu Liu et al. “Survive Under High Churn in Structured P2P Systems: Evaluation and Strategy”. In: *Proceedings of the 6th International Conference on Computational Science - Volume Part IV*. ICCS’06. Reading, UK: Springer-Verlag, 2006, pp. 404–411. ISBN: 3-540-34385-7, 978-3-540-34385-1. DOI: 10.1007/11758549_58. URL: http://dx.doi.org/10.1007/11758549_58.

[15] Philip Matthews, Jonathan Rosenberg, and Rohan Mahy. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. IETF RFC 5766. Oct. 2015. DOI: 10.17487/rfc5766. URL: https://rfc-editor.org/rfc/rfc5766.txt.

[16] Philip Matthews et al. *Session Traversal Utilities for NAT (STUN)*. IETF RFC 5389. Oct. 2015. DOI: 10.17487/rfc5389. URL: https://rfc-editor.org/rfc/rfc5389.txt.

[17] Colin Perkins, Joerg Ott, and Magnus Westerlund. *Web Real-Time Communication (WebRTC): Media Transport and Use of RTP*. Internet-Draft draft-ietf-rtcweb-rtp-usage-26. Work in Progress. Internet Engineering Task Force, Mar. 2016. 46 pp. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage-26.

[18] Eric Rescorla. *Bug 852665 - Report WebRTC transport termination (e.g. iceConnectionState=disconnected)*. Mar. 19, 2013. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=852665.

[19] Eric Rescorla. *WebRTC Security Architecture*. Internet-Draft draft-ietf-rtcweb-security-arch-11. Work in Progress. Internet Engineering Task Force, Oct. 2015. 43 pp. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-11.

[20] Jonathan Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*. IETF RFC 5245. Oct. 2015. DOI: 10.17487/rfc5245. URL: https://rfc-editor.org/rfc/rfc5245.txt.

[21] Marc Stevens, Pierre Karpman, and Thomas Peyrin. *Freestart collision for full SHA-1*. Cryptology ePrint Archive, Report 2015/967. http://eprint.iacr.org/. 2015.

[22] Ion Stoica et al. “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”. In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’01. San Diego, California, USA: ACM, 2001, pp. 149–160. ISBN: 1-58113-411-8. DOI: 10.1145/383059.383071. URL: http://doi.acm.org/10.1145/383059.383071.

[23] Unknown. *Comparison with Tor - I2P*. Mar. 23, 2016. URL: https://geti2p.net/en/comparison/tor.

[24] Unknown. *Garlic Routing and ”Garlic” Terminology - I2P*. Mar. 1, 2014. URL: https://geti2p.net/en/docs/how/garlic-routing.

[25] Unknown. *Tech Intro - I2P*. Mar. 23, 2016. URL: https://geti2p.net/en/docs/how/tech-intro.

[26] Unknown. *The Network Database - I2P*. Nov. 1, 2010. URL: https://geti2p.net/en/docs/how/threat-model.

[27] Unknown. *The Network Database - I2P*. Feb. 1, 2016. URL: https://geti2p.net/en/docs/how/network-database.

[28] C. Vogt, M.J. Werner, and T.C. Schmidt. “Leveraging WebRTC for P2P content distribution in web browsers”. In: *Network Protocols (ICNP), 2013 21st IEEE International Conference on*. 2013, pp. 1–2. DOI: 10.1109/ICNP.2013.6733637.