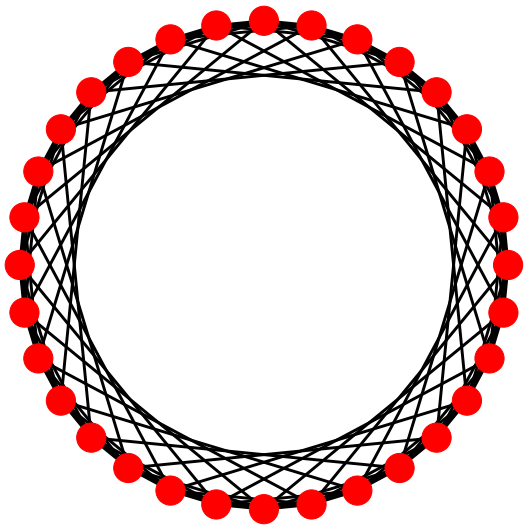
Nathaniel J. McAleese

Browser-hosted P2P Resource Discovery

Computer Science Tripos – Part II

Gonville & Caius College

May 16, 2017



Proforma

Name:	**Nathaniel McAleese-Park**
College:	**Gonville & Caius College**
Project Title:	**Browser-hosted P2P Resource Discovery**
Examination:	**Computer Science Tripos – Part II, July 2017**
Word Count:	**11, 982** [1]
Project Originator:	Nathaniel McAleese-Park
Supervisor:	Dr. J. Singh

Original Aims of the Project

To design and implement a peer to peer resource discovery system that would allow users to join and contribute using their web browser. The project aimed to achieve high availability even as users join and leave the network regularly.

Work Completed

Keyword search was selected as the means of resource discovery, and a distributed search system was built on top of a structured overlay network. A routing protocol, search protocol and replication mechanism were implemented in Typescript, using WebRTC for P2P connections. Tests using a realistic search dataset on a simulated network show that the system operates correctly under stable conditions and is tolerant of node churn. This fulfilled all the requirements of the project and met all the success criteria outlined in the project proposal.

[1] Word count from TeXcount web service [4]

Special Difficulties

None.

Declaration

I, Nathaniel McAleese-Park of Gonvile & Caius, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date 11/5/2017

Contents

List of Figures

Acknowledgements

Friends and family who provided tea and cake know already that they are loved, but perhaps not how much that contributed to this work. More practical thanks go to Jat Singh, whose detailed insights were invaluable, and to all others unfortunate enough to read a draft. Martin Richard's LaTeX template was also extremely useful.

Chapter 1

Introduction

This project aims to make peer to peer (P2P) systems easier to develop and more accessible by creating a distributed resource discovery system that works in the web browser. No additional software is required, and so users may join the network by simply navigating to a web page. After this initial "bootstrap", access to the P2P network persists even if that web server hosting the original page fails.

The rest of this chapter outlines the motivations for the project in more detail, and gives an overview of its goals and related work.

1.1 Motivation

Numerous arguments have been made for the benefits of peer to peer systems.

- Low barrier to deployment - once developed, a P2P system can be deployed at minimal cost. This motivated the use of peer to peer technology in the early development of Skype [10] and Spotify [33], for example.
- Organic growth - Resources are contributed to a peer to peer system by the users, ideally allowing for rapid scaling without the original developers having to make infrastructure changes, such as upgrading servers.
- Fault tolerance - ideally no peer is critical to the system, and thus there is no central point of failure. Nodes may join and leave without affecting service quality for other users.
- Privacy preservation - peer to peer networks can prevent single organisations from accumulating an invasively large amount of data.

Despite these advantages, the popularity of peer to peer systems has not grown proportionally to usage of the internet in general. This is partially due to greater complexity

of joining P2P systems, which require the installation of specialist software and often do not work for users behind NAT.

This can be contrasted with use of the web browser. Huge numbers of users frequently interact with numerous web applications and in general trust the browser's security model. Lay users do not want to download additional software due to security concerns and lack of technical knowledge, and thus the browser is by far the most used piece of software on the average person's pc.

To make P2P networking more accessible, it therefore makes sense to develop applications for the browser.

Very few web applications, however, currently capture the advantages of peer to peer networking. Indeed, until recently, the web browser was entirely designed around a client-server model. This is because, for security reasons, the browser only offers limited access to the host system. The recent adoption of WebRTC [28], a standard for direct peer to peer communication between web browsers, has begun to change this, but the semantics of WebRTC connections are distinct from those of more traditional TCP/UDP sockets. This is because they require out-of-band negotiation before a connection can be established, which complicates their use.

As a result, many systems that use WebRTC maintain several points of centralisation. One particularly crucial element that is still centralised can be generically characterised as "resource discovery". In P2P systems, peers often need to discover resources that satisfy some set of criteria; for example documents on a particular topic, other peers interested in the same file, or any other resource the network may offer. Indeed resource discovery can encompass both human users seeking to satisfy an "information need", and automated requests by software to determine machines or services that may be interacted with.

1.2 The Project

In order to give web developers greater access to the benefits of decentralisation, this project develops a distributed resource discovery system that works in the browser. It allows peers to store and retrieve identifiers from a distributed index that persists in the face of peer failure.

An example site demonstrates the utility of the system by using it to implement a decentralised file sharing scheme. This demo uses the distributed resource discovery system alongside an existing P2P system called WebTorrent [6]. The architecture of the demo is outlined in more detail in section 3.10.

The example file sharing system uses this distributed resource discovery mechanism, so both files and the search index are distributed. Thus anyone who has previously accessed the site **may still upload, search for, and download files even if all of the developer's servers fail**.

Using a distributed system to discover files and peers also has further advantages:

- Multiple "bootstrap" pages may offer connections to the same network. This may be useful, for example, to evade censorship by DNS blocking.
- The users' data is distributed amongst the users, and does not need to be stored by a central administering authority.

1.3 Related Work

A great deal of work has been done on the design of peer to peer networks, distributed hash tables and distributed resource discovery systems. Gnutella [43] and Napster [2] formed the first wave of peer to peer systems, using "unstructured" networks and offering search functionality.

CHORD [48], Kademlia [38] and Pastry [45] offer designs for distributed hash tables. These create a decentralised key-value store service for peers that join the network, and all three implement key based routing.

Several efforts have been made to build upon the key based routing primitive offered by these distributed hash table (DHT) systems in order to implement search or resource discovery. Both Sword and Mercury are systems that use the KBR primitive to implement multi-attribute range queries. These allow peers to discover resources such as available peer computers based on, for example, the requirement that their cpu usage and available memory both fall within some desired range [9, 14]. These systems are both quite complex. Mercury relies on KBR as an established operation to create multiple routing rings, one for each attribute. This necessitates knowing a priori what schema will be appropriate for the application, and also increases the cost of sharing one network between multiple applications because the cost per peer scales linearly with the number of attributes that may be searched over.

The authors of Mercury note that the "highly desirable flexibility offered by keyword-based lookups" can also be used for resource discovery. Keyword search has been implemented as an extension to key based routing a number of times [29, 36, 41, 35].

None of the above systems have been implemented for the browser, and they do not consider the issues presented by the popularity of NAT, which can complicate peer to peer connections.

The most popular distributed application available from the browser is WebTorrent [6]. This allows users to download large files from other peers, and has formed the basis of a distributed CDN (content distribution network) that allows site owners to save on bandwidth costs by having their end users contribute to hosting a site's media. The WebTorrent project does not, however, currently allow browsers to begin downloading files without access to a centralised server [7]

Chapter 2

Preparation

Implementing the distributed system required the use of software engineering techniques as well as a detailed understanding of previous P2P systems. This chapter outlines the work that was done before implementation began to ensure that the project was successful.

2.1 Backups and Version Control

Two mechanisms were used to prevent data loss. The cloud storage system "Dropbox" was used in order to backup the current state of my project folder and decrease the chances of catastrophic loss. As development proceeded, changes to the source were also checked into the version control system "git", and pushed to a private repository on BitBucket.

2.2 Choice of language

All popular web browsers implement an interpreter for a language called Javascript. The original language was designed in ten days [47], and it is notorious for presenting problems for developers. Frequently cited issues are the lack of native integers (all numbers are floating point), and dynamic typing that is difficult to reason about. For example:

```
> {} + []
0
> [] + {}
'[object Object]'
> [] + []
''
```

To mitigate these and other issues I elected to use Typescript [5]. This is a syntactic superset of Javascript that offers static analysis features and some small extensions to the language. It can be transpiled to code that will run in all popular browsers. It allowed me to use some of the techniques taught in "Foundations of Computer Programming", such as algebraic data types, to help prevent bugs. I had very little prior experience with Typescript, so a portion of the preparation time for the project was spent familiarising myself with both it and Javascript.

2.3 Development Methodology

The undergraduate software engineering course makes clear the importance of a clear software development process. Because the proposal required laying out success criteria for the project and milestones to be completed in advance this naturally pushed development towards a waterfall-style methodology. In the first weeks, the initial requirements laid out in the proposal were fleshed out, and an initial design was conceived. However at the end of each 2 week period between milestones I considered progress so far and whether the goals of the projects may have to be changed in order to minimise project risk.

2.4 Requirements Analysis

2.4.1 Use Cases

There are numerous potential use cases for a resource discovery system. Consider that interactive sites might need to allow users to discover:

- named users
- named communication channels
- machines satisfying some arbitrary requirement
- documents on a topic

However to prevent "scope creep" two particular use cases were focused on in order to determine the functional requirements of the system:

- file search
- browser support for "trackerless torrents"

These two uses were selected as they represent both a human user's information need (file search) and an automated request. They also inter-operate neatly to provide a

demonstration site that illustrates the utility of the system. The motivations for and requirements of each are outlined in more detail below.

2.4.2 File search

File search has been a feature of peer to peer networks since the initial development of Gnutella and Napster. Users in a file search system have two primary interactions - they want to share a (potentially large) file with other users, or they want to search the available files. In particular, users tend to engage in either "known item search" or precise information seeking search as opposed to topical search. Thus a file search system should allow users to search for available files and provide some means of retrieving them.

2.4.3 BitTorrent & Trackerless Torrents

BitTorrent is a protocol that allows a "swarm" of peers to share large files. It allows the bandwidth costs of serving the files to be distributed amongst users that are interested in the file. A "torrent file", or "torrent" refers to the metadata of a particular file that is being shared with the system. It contains, amongst other things, the file's "info-hash", which is a cryptographic hash of the file's contents.

In order for the protocol to operate, peers must be able to discover others that are interested in the same file. This is accomplished through a "tracker". Initially trackers were implemented as a centralised server that allowed a peer to issue HTTP requests to discover other peers. In order to reduce service disruption due to tracker failure, many bittorrent clients adopted a "trackerless" approach. In this model, each peer joins a Kademlia based distributed hash table (DHT). To register their interest in a torrent, a peer pushes their contact information into the distributed hash table for the torrent's info-hash. When they wish to determine which other peers are interested in a particular torrent, they look up that same hash. This allows them to discover other peers from only the info-hash (which is easily communicated and does not change).

WebTorrent does not currently implement a DHT for browser peers. A DHT implementation is a frequently requested feature [8], but the developers have noted that producing a working implementation of the Kademlia DHT under the restrictions of WebRTC is non-trivial.

In order for the resource discovery system to serve as a bittorrent tracker it must allow peers to register their interest in a torrent. It must also allow them to determine which other peers have registered an interest, and provide some mechanism ensure that at least some of the peers returned are still active, and that the number of returned peers does not grow arbitrarily over time.

From these uses cases, we can determine that the system should:

- Have a search mechanism which is appropriate for both automated lookup and precise information seeking search by users.
- Allow peers to store resource identifiers so they may then be discovered.
- Ensure that stale data does not persist in the network (for example, peers that are no longer available).

The rest of this chapter details the early architectural decisions made to ensure that the system could satisfy these core goals, before isolating the more specific functional requirements determined from this design. To motivate these decisions, it outlines the two primary approaches to peer to peer networks, along with the restrictions of WebRTC and the browser environment.

2.5 Considerations for WebRTC

The design of overlay networks traditionally assumes that the underlay can route a message from one peer to any other. For example, the original design of CHORD assumes that it is sufficient to store an identifier, an IP address and a port for each peer, and that messages may then be sent over UDP.

The interface available to code running the browser is more restrictive. A WebRTC connection must be negotiated between the two browsers before data can be sent between them. This involves exchanging "signalling" data between the peers. Even before considering the nature of this exchange in more detail (in section 3.3) we can note that this will introduce new considerations into the design of an overlay network.

Working within these restrictions is worthwhile for two reasons. Firstly, some sort of signalling exchange is necessary to accommodate users behind NAT (see section 3.3). Secondly, a solution that uses the currently available API is much easier to deploy. For example, the demo site produced for this project immediately worked on my Android phone over the cellular network; something that would have been non-trivial to achieve if modification of the browser was required.

In order to surmount these problems it is necessary to introduce two distinct types of peer. Publicly addressable peers are users that are not behind NAT. This can be identified by the information included in their signalling requests [3], after which they should be prompted to acquire a modified copy of the application code that allows incoming connections. This could be implemented as a browser extension, or as a separate desktop application.

We should expect publicly addressable peers to be a relatively scarce resource, as they require both additional user effort and an appropriate IP address. I therefore sought to ensure that the additional load placed on these peers is minimal (as excessive bandwidth consumption is likely to discourage participation), and further that the network is as resistant to their failure as possible.

2.6 Approaches to P2P Networks

There are two primary approaches to the design of P2P networks, structured and unstructured. This section explains the advantages and disadvantages of each, and why a structured approach was chosen for the project.

2.6.1 Unstructured P2P Networks

Gnutella and Gia [43, 23] are classic examples of unstructured peer to peer networks. They use, respectively, flooding and random walks in order to route messages and respond to user queries. In these systems, each peer has a local database of file identifiers (indicating where a file may be fetched from) and associated metadata (such as the file name). Upon receiving a query, a peer searches its local database. If a result is found, it sends it back to the originating peer. Otherwise, it forwards it either to another random peer (random walk), or all the peers to which it is connected (flooding). Results are also cached by nodes that forward them, so popular files become more widely replicated in the network and therefore may be found in fewer hops [18, 23].

As has been said before, this approach is very effective at finding "hay". It is not, however, appropriate for finding "needles" [23]. In other words, it is useful for finding popular files or resources. The problem is that in the flooding model, rare resources create a lot of network traffic, and in the random walk model searches for rare resources are either not guaranteed to return results (because there is some TTL associated with messages that eventually expires) or have large or unbounded latencies (as messages are sent randomly around the network without visiting the node that holds the resource) [18, 23].

Neither of these situations are desirable. Unstructured P2P networks also increase the complexity of storing dynamically updated resources. This is because a file identifier may be stored by any peer, and thus an update could not guarantee that all peers storing the resource were updated without being sent to every peer in the network.

Finally, unstructured P2P networks do not offer the ability to efficiently route a message to a specific peer. This would have greatly complicated the implementation of the signalling exchange required by WebRTC [18].

2.6.2 Structured P2P Networks

In a structured peer to peer network, the nodes seek to maintain a specific network topology. This allows messages to be routed between peers in a bounded number of hops under stable conditions, and requires that the peers only maintain a small amount of routing state. For example, both CHORD and Kademlia offer a bound on both the number of hops between any two peers and the amount of state that they must store that is logarithmic in the size of the network. This has allowed Kademlia networks in particular to scale to millions of nodes. Most structured P2P networks offer a key based routing mechanism. In such a system nodes take their identifiers from a "flat" keyspace. A frequently used example is an arbitrary 160 bit number. The system then offers the primitive `send(msg, n)` that routes a message to node in the network with the id closest to `n`. This has been used to implement numerous other higher level systems, but of particular importance is that it allows one peer to messages to any other efficiently, as well as to an arbitrary location. Queries in structured overlays are usually more specific than in unstructured networks - for example, a distributed hash table offers only the exact lookup of results for a particular key.

2.7 Results of preparation

Having considered the available options, I determined that a structured network was appropriate for the project. This is because a structured network has two key advantages for this system:

- It allows the system to efficiently route messages to a particular node, allowing for in-network exchanges of signalling data for new connections. This reduces the load on the publicly addressable peers, and allows for a greater degree of decentralisation.
- It allows for the efficient discovery of rare data, such as the newly created torrents that do not (yet) have many interested peers.

My reading also suggested that, as noted in the proposal, an exact-match keyword search interface would be extremely useful without entailing the prohibitive complexity of structured query systems such as SWORD or Mercury.

2.8 Functional Requirements

These considerations allowed me to determine the functional requirements of the scheme. The system should:

- Allow peers to store resources associated with a set of keywords.

- Allow peers to specify a lifetime for stored resources, up to some maximum. Thus stale resources do not persist in the network indefinitely. Resources may have their lifetime extended by republication[1].
- Support conjunctive queries to discover resources. More specifically, a user or client application should be able to execute a search by providing a set of keywords. If a resource identifier stored in the distributed index is associated with any superset of that query set, it should be returned by the network. If there is no such resource identifier, then an empty result set should be returned.

The third of these functional criteria must be met in stable networks with connected nodes. However, we can note that fufilling it in a network with random link failures and unbounded latencies would violate the CAP theorem [16]. Thus this project aims to offer eventual consistency (described in section 3.6.2), and the evaluation seeks to determine how frequently incorrect results are returned due to node churn.

2.9 Non-functional requirements

The system should:

- Accommodate large numbers of nodes by ensuring that local state and path lengths grow sub-linearly with the number of peers in the network.
- Ensure that the lifetime of stored resource identifiers exceeds the average uptime of a peer in the network.
- Allow peers that have a cached copy of the code to join the network even if all the developer's machines have failed.

[1] This is the approach taken by DHash and Kademlia for maintaining keys.

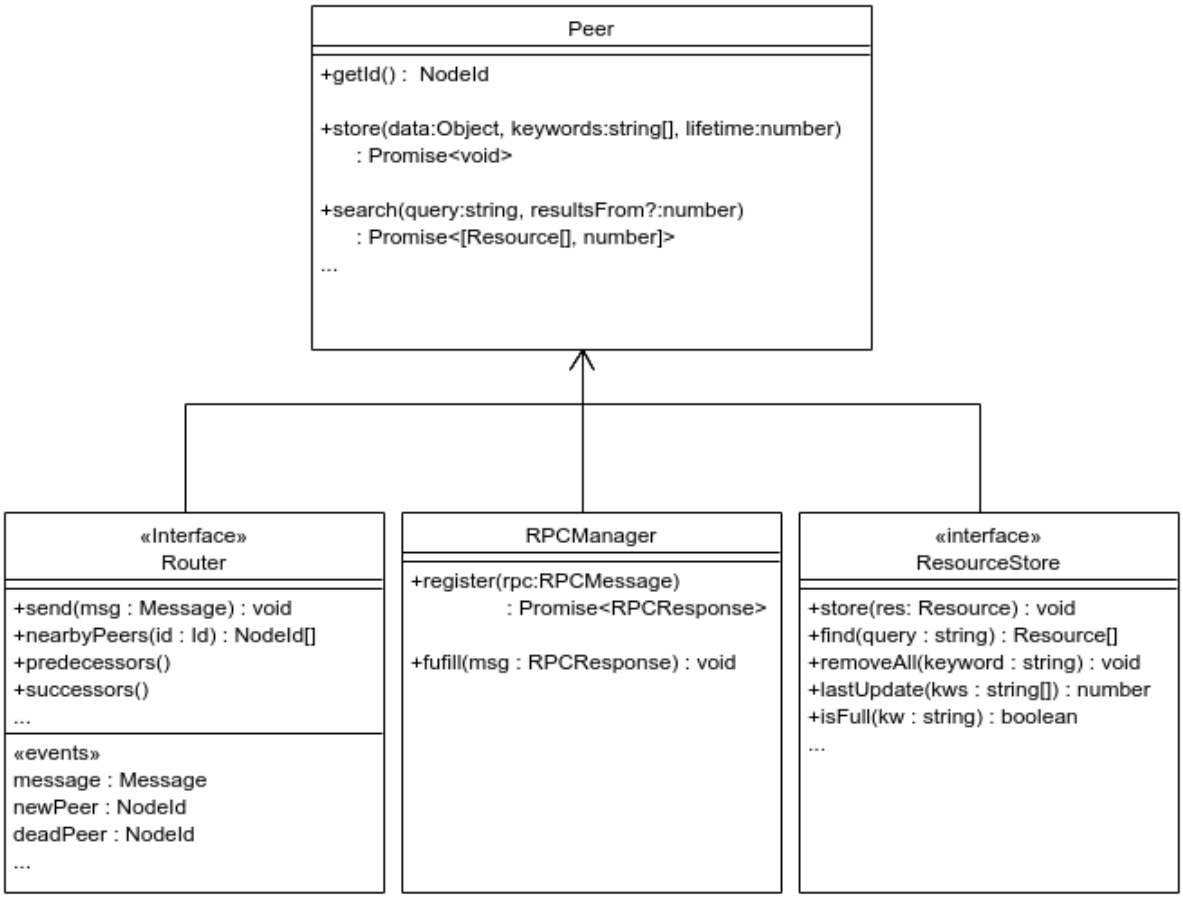


Figure 2.1: The main classes of the project.

Chapter 3

Implementation

This chapter outlines the work done in implementing the system, and includes a more detailed description of the algorithms involved. Some practical details, such as the build system used to bundle code, are relegated to the appendix (section A). As illustrated in figure 2.1, the project was divided into 4 core classes. A high level Peer class uses a ResourceStore, a Router and an RPCManager to handle the implementation of the protocol. The considerations of each component are in the following sections.

3.1 Testing

The final system has a number of inter-reliant components. For example, it is necessary to route signalling data correctly in order to establish the connections over which signalling data may be routed. Under such conditions, maintaining modularity and information hiding are important to ensure that a system does not become unmanageably complex.

To this end, I aimed to ensure that the APIs offered by each class were small and well tested. This also enabled several periods of development to proceed using mocked forms of other components - for example, a test router that used a centralised mechanism to deliver all messages.

Nonetheless, I found that debugging problems in this distributed system was non-trivial and wrote several tools to aid with the manual inspection of log data. These included, for example, a tool to graphically trace the routes of, and routing decisions made on, a particular RPC, as well as tools to visualise the state of a peer's routing table over time.

3.2 Messages

Peers communicate by sending messages. The type system was used to ensure that all messages were well formed when constructed. While in memory they are stored as Javascript objects, and a serialization protocol was used to serialize these messages to and from an appropriate binary format for transmission over the network. A base message type was used to ensure that all messages had an origin, a destination and a unique message id, all other message types extended it.

3.3 WebRTC

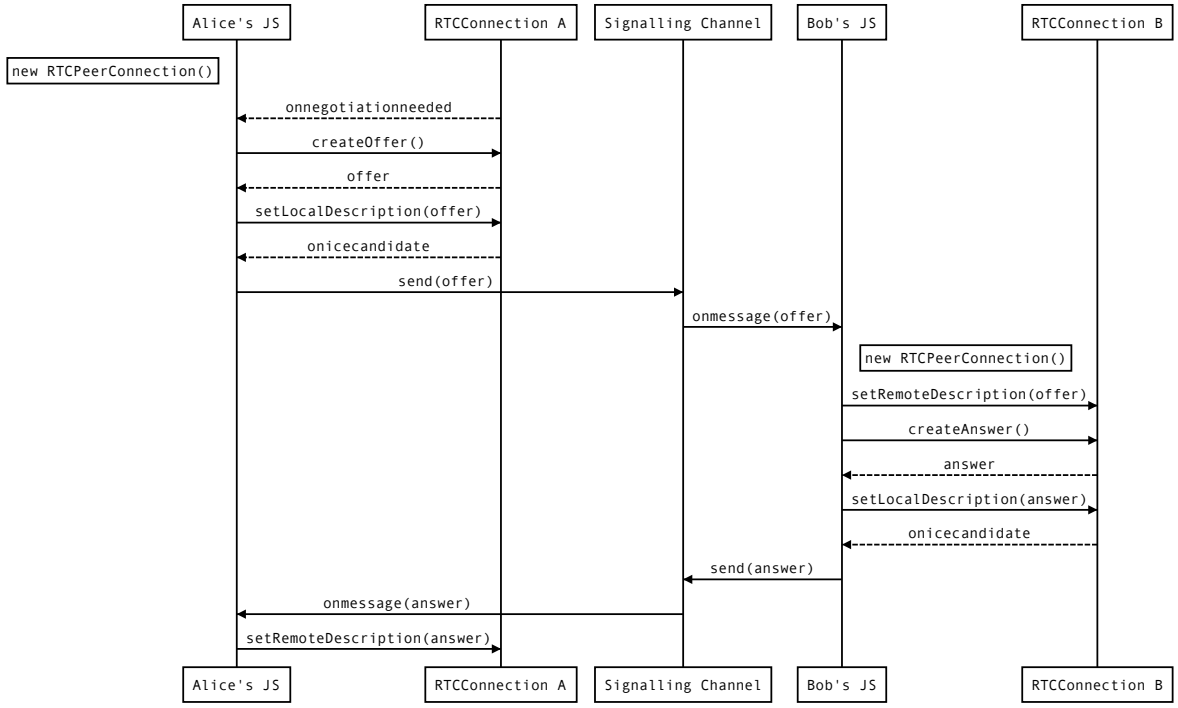


Figure 3.1: A sequence diagram of a WebRTC signalling exchange. Solid lines indicate method calls, dotted lines indicate events emitted asynchronously by the connection object.

3.3.1 WebRTC Signalling, ICE, STUN and NAT

Figure 3.1 shows a sequence diagram demonstrating how to establish a WebRTC connection. This section explains the steps involved and why they are required.

Network address translation (NAT) is a widely deployed solution to IPv4 address shortage. It allocates a single IPv4 address to an entire private network, and maintains a translation table. NAT operates by sitting between the private local network and

the internet at large. All datagrams that flow through the NAT are modified, and in particular datagrams containing TCP and UDP packets have their source address modified to be the IP of the NAT server. The origin port of the packet is then modified so as to unambiguously identify the origin of the message. Thus response packets to the NAT on that port may be altered to direct to the original source on the private network.

This complicates the deployment of peer to peer networks because many devices are no longer addressable until you know what IP and port a NAT is mapping them to. One means of traversing NAT is through use of a STUN[1] server [50]. This is a publicly addressable server that responds to incoming requests with a response containing the IP and port from which the packet originated.

Because NAT behaviour varies, it is not always possible to establish a connection this way. However developers of VoIP products suggest STUN succeeds for their customers around 80% of the time.

The "onicecandidate" events in figure 3.1 refer to candidates for Interactive Connectivity Establishment (ICE) [44]. This is a protocol that tries to establish connectivity between two peers by a number of means, including direct connection after a STUN server is contacted to discover their external IP. The availability of STUN servers will be assumed for the rest of the project - any publicly addressable peer can serve as one to the network, and the cost of responding to STUN requests is negligible compared to the rest of the network's functionality.

The offer and answer exchanges also contain the other information required to set up a WebRTC connection. The transport layer protocol used by the data channel is the stream control transmission protocol (SCTP). This an alternative to TCP and UDP that offers congestion control and configurable reliability.

The WebRTC also requires transport layer security between the endpoints. Datagram transport layer security (DTLS) is used, and a sha256 hash of the certificates included in the signalling data. If the correct hash is not present in the answer then the connection is rejected.

Thus in order to establish a WebRTC connection (as depicted in figure 3.1) a peer must generate an offer, wait for ICE candidates (e.g. its public IP to be returned from a STUN server) and send the offer to the peer to which it is connecting through some signalling channel. The receiving peer must then accept the offer, wait for its public IP and then generate an appropriate answer, before sending it back to the originator through the signalling channel.

The key takeaways for the project are that:

- Connections require the exchange of signalling data by some means before connectivity is established.

[1] Session Traversal Utilities for NAT

- Once established, transport is reliable.

The management of these connections was implemented first. This was done by creating a mock Router class that forwarded all traffic through a central server. The code to manage the exchange was then implemented using this mock router to directly deliver all messages sent by the peers. This allowed for the following interface to be created on the Peer class:

```
getP2PConnection : (partner:NodeId) => Promise<P2PConnection>
```

Which represented a substantial abstraction over the full exchange shown in figure 3.1.

3.3.2 RPCManager

Implementation of the protocol required a remote procedure call (RPC) scheme, to allow nodes to send queries and handle the results for example.

New RPCs were added by creating message types that extend the base RPCMessage type which includes the required fields. A handler was then also implemented on the Peer for each new RPCMessage type.

It would clearly be sub-optimal to block on every RPC. Thus an RPC manager was implemented that maintains a least-recently used cache of outstanding RPC closures, and fulfills them with the results that reach the peer or times them out if they take too long.

Initially these timeouts used exponential averages of the mean and variance of RPC times to set the timeout to be $\mu + 3\sigma$, but I quickly realised this was inappropriate due to the varying number of hops taken by messages in the network. It was then replaced by setting the timeout to 3 seconds or twice the longest delay of any of the last 100 RPCs, whichever is greater.

3.4 Routing

3.4.1 CHORD

CHORD is the most widely cited design for a key based routing scheme, and is closely related to the mechanism I implemented. In CHORD, each node is given a unique numeric id from some fixed id space of ids in $[0, N]$, and these ids are arranged in a virtual ring. The successor of a node with id n, that follows it in the ring, is defined to be the node with smallest id greater than n, modulo N.

Each node stores the following routing information: a successor list, containing its k successors, and a *finger table* containing nodes at exponentially increasing distances around the ring. In particular, the ith entry in the finger table is the first node with an id that exceeds $n + 2^{i-1} \mod N$.

If nodes are uniformly distributed over the address space then this routing scheme guarantees a path between any two nodes in the stable network in $O(\log n)$ hops. It also only requires $O(\log n)$ space for the routing table at each node. Intuitively this is because each hop towards the target node should at least half the remaining distance; for a more formal proof see [48].

3.4.2 S-CHORD

In order to reduce the number of connections maintained per peer, and to better utilise the duplex nature of WebRTC connections, I implemented S-CHORD, a variation of CHORD described in [39]. This defines a notion of distance between two ids as the distance around the identifier ring, clockwise or anti-clockwise, whichever is less. When forwarding a message, the next hop is selected to be the closest node to the destination.

The finger tables in S-CHORD are also different. Intuitively, we divide the identifier space into two halves for each node, specifically those ids to which the clockwise and anticlockwise distance is less, respectively. More formally, the ith entry of the finger table of node i is defined as:

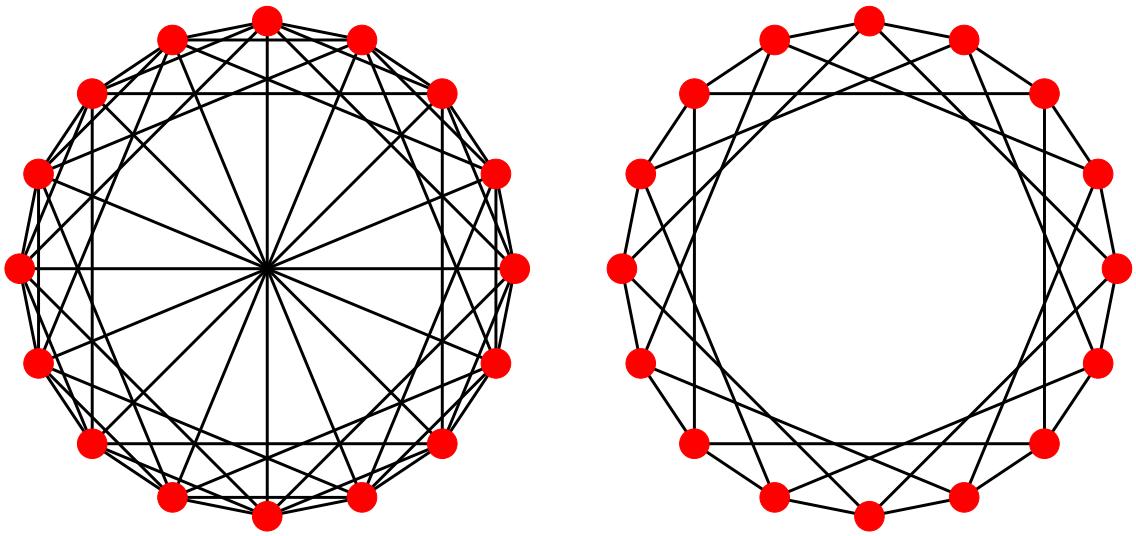
$$n.\text{finger}[i] = \begin{cases} \text{successor}(n + 4^{i-1} \mod N) & i \in [1, m] \\ \text{predecessor}(n - 4^{2m-i} \mod N) & i \in [m+1, 2m] \end{cases}$$

Where $m = \lceil \log_4 N \rceil$

The asymptotic behaviour of this approach is the same as that of CHORD; offering both a worst case path length and maximum number of fingers that are logarithmic in the number of nodes in the network.

It does, however, offer some advantages. In particular, this approach reduces the cost of notifying the network that you have left, and reduces the number of active connections that each node must maintain. This is due to the symmetry of the routing tables - using CHORD, each node would maintain a set of connections to its finger nodes both for those nodes for which it was a finger, as well as those nodes in its finger table. In S-CHORD, there is substantial overlap between these two sets, reducing the total number of connections per node [2]. Figure 3.2 illustrates the difference.

[2] As the number of peers grows large, the improvement tends towards a factor of two.



(a) A CHORD network of 16 nodes. 7 connection per peer.

(b) An S-CHORD network of 16 nodes. 4 connections per peer.

Figure 3.2: S-CHORD reduces the number of connections per node at no additional routing cost.

3.4.3 Router Implementation

An open source implementation of a balanced binary search tree [1] was extended to allow for fast lookups of nodes by proximity. To find the closest node to a specified id, we can use the tree to quickly find the stored nodes that immediately precede and follow it. We can then compare their distances to to determine which is closer. This can be extended to quickly find a sorted list of nodes by distance from a target key.

The routing approach deployed in this project had to be adapted to the restrictions of WebRTC. Because connections must be negotiated before they are used (and not all pairs of nodes can form a connection due to NAT issues), the message routing must satisfy certain properties

- Routing must be recursive, because the overheads of negotiating a new connection to every contacted node would be prohibitively high.
- The routing mechanism must be able to find paths through the network even when greedily routing towards the next nearest node would fail.

The second requirement is necessitated both by the potential for NAT connectivity issues and because nodes will often not be connected to their immediate neighbours. For example, when a new node joins the network it will generate an id and connect to a bootstrap node. When it sends its first RPC to discover appropriate neighbours and begin filling its routing table, the response must be routed back to this peer despite the fact that it is "out of place" in the network, e.g. not connected to either of its predecessor or successor.

Two techniques were applied to surmount this problem. Firstly, NACKs [3] and retries were added to the recursive routing methods, duplicating the routing approach of Freenet [24]. A node will forward a message to the best next hop that it has seen, and retry if that node NACKs. If all its next hops fail it will NACK the message itself. This is equivalent to a greedy depth first search, and with a sufficiently large number of retries will always discover a path if one exists. It is, however, very expensive.

To reduce the cost, a node will also store the last origin of every node in its routing table. This set is populated by inspecting the fields of messages that pass through the node. When forwarding a message, a node inspects its routing tables for the closest node that it has seen to the destination, and selects the next hop for the message as the last origin for that node. If it is connected to no closer node, and has not seen a message from any closer node then `nextHop` returns null.

There are also two types of messages that must be routed by the network - exact messages, such as an RPC response that needs to reach the original sender of the RPC, and inexact messages that need to reach the closest node to their destination, such as as looking up a key. In the second case, if greedy routing fails then we allow any peer that is within k of the destination to respond, where k is the replication parameter of the higher level system.

```
onmessage = (msg, prevHop) => {
  let nextHop = nextHopFor(msg, prevHop)
  if (nextHop) {
    forward(msg, nextHop)
  } else if (!needsExact(msg) && withinKOf(msg.destination)) {
    handle(msg)
  } else {
    NACK(msg)
  }
}
```

Other approaches were also considered, including those of Virtual Ring Routing [19], which allows for key based routing on an arbitrary network topology, as well as more traditional path vector and link state approaches. These do not, however, share the logarithmic scaling of the P2P approaches; and because the topology of the overlay network adapts towards the S-CHORD ring it is only necessary to accommodate a comparatively small amount of non-greedy traffic. Thus implementing the amount of routing overhead required by these alternative schemes seemed unnecessary.

[3] negative acknowledgements

3.5 Keyword Search

There have been several approaches suggested to implementing keyword search on top of a KBR overlay. All rely on implementing some form of inverted index. An inverted index maps a keyword to a list of the resources associated with that keyword, and allows conjunctive queries to be answered by using the index to look up the set of resources associated with each keyword in the query and taking their intersection.

Reynolds [41] discusses how such an index should be partitioned across the nodes in a network. Horizontal partitioning divides the resources amongst the nodes by the resource identifiers. This has the advantage that the number of nodes that must be contacted for updates or insertions is independent of the number of keywords that a resource is associated with. Vertical partitioning, by contrast, stores all the resources associated with a particular keyword on the node with the closest id. This reduces the cost of queries in the system. In the suggested use cases we expect queries to be more numerous, so a vertical partitioning scheme is used.

More specifically, suppose we have a resource with id r that has a set of associated keywords K. In the vertical partitioning scheme we store this resource id in the network once for each $w \in K$, at the node which has the closest id to the hash of w.

In [41] queries are then serviced by sending an RPC to the node which has the id closest to the first keyword. If the query is multi-word it then determines which resources should be returned through an exchange of Bloom filters [4] with node closest to the hash of the next keyword. This is proposed as a means of reducing the amount of traffic associated with multi-word queries. I chose to implement a different approach that reduces the amount of traffic required further at the cost of additional storage. In addition to storing all the resource identifiers associated with a particular keyword, the list of keywords for each is stored. Thus if a resource is associated with n keywords this increases the cost of storing the identifier in the network from $O(n)$ to $O(n^2)$. However I would suggest that for useful values of n this is unlikely to be an overwhelming cost. Tweets, newspaper headlines and classified titles all, for example, tend to be 20 words or less.

This does makes full text search of longer documents infeasible, but has several advantages. It is simpler than a multi-peer exchange of Bloom filters. It also allows every query, regardless of the number of keywords, to be serviced with a single RPC. This reduces query latency, which is very important to human users of search systems [17]. We can also route the query towards the keyword hash which is closest to the originating peer, which further reduces latency and allows more queries to be serviced locally with no RPCs at all.

[4] Bloom filters are discussed in more detail in section 3.6.4

3.5.1 Issues with vertical partitioning

Vertical partitioning in general, however, has one primary problem. If the distribution of keys in the DHT is not uniform over the id space, then the load on each peer is unlikely to be uniform. The distribution of word usage in natural language is not uniform at all. Indeed, it is Zipf distributed, with the most frequently used word being used twice as often as the second most frequent and so on. This has the potential to induce a Zipf distributed load on the peers, which is not desirable. Consider, for example, that one peer would have to store every identifier associated with the word "the". Several solutions have been proposed, one of which is "Keyword Fusion". This uses a gossip protocol to exchange a data structure containing a list of words that are too frequently used to be directly stored in the inverted index and instead "fuses" these keywords with the others with which they occur.

It has been argued, and intuition would suggest, that the most frequent terms (and those that would cause the most problems for a naive implementation, such as "the") are not particularly informative in searches. Indeed, if the system attempted to accommodate queries such as "the and", it seems likely that they would greatly increase communication cost without providing much additional benefit to the user at all.

I therefore took two approaches to preventing the worst of these issues, with a mind to the proposed use cases.

3.5.2 Active load balancing

Peers will store up to a maximum number of results for each keyword, after which they will mark it as full. On each new store, they will then discard the oldest resource that they stored for that keyword. When responding to a request for a too-frequent keyword, they will mark the response. This allows other keywords to automatically be tried from the query. If any of the keywords in the query is not too-frequent, then all the results for the query will be returned. If all are too frequent, then recent (and hopefully useful) results will be returned, and this can be indicated to the user who may choose to reformulate the query to be more specific if the recent results were not sufficient. A local copy is also kept of terms that responses indicate as too-frequent. This reduces query load on the peers hosting those terms[5].

This limits the storage and replication costs for popular terms, without overly hindering the intended use cases.

[5]This is maintained as soft state with an associated expiration time.

3.5.3 Hard-coded load balancing

Whilst considering load balancing options, I also decided on another practical approach: distribute a list of the 100 most common words in English with the application, ignore these in storage requests and alert the user if they are included in queries. This has obvious shortcomings. The first is that it is not an effective technique if the distribution of keywords is non-uniform and not similar to the distribution of the use of words in general English. This will clearly be true for foreign languages, but will also potentially be true for English users because the domain of internet search requests is distinct from that of literature and news articles (from which the frequencies were gathered). For a practical example, we might expect the word "video" to occur much more frequently in searches than in the corpus used to generate the frequency statistics.

The upside of this approach is that it is made more efficient by the same Zipf distribution that was originally problematic. The list of overly frequent words can be small, and still substantially reduce the number of resources associated with the most frequent term. For example, in English the most frequently used word occurs around 65 times more frequently than the 100th, so we might expect around a 65x reduction in the storage required at the node hosting the most popular word. This hard-coding could also account for likely bug results, such as the empty string, and could be adapted to any other use cases.

The problem that this does not address is how to handle intentional queries with too many results. To handle these, query messages include an integer `resultsFrom`, and results on the peer servicing the query are ranked by Jaccard similarity (with ties broken by recency), and some small number of returned from `resultsFrom` onwards.

3.6 Replication

In order to satisfy the requirement that resources stored by the network are not lost due to node joins and leaves, it was necessary to implement a replication system.

3.6.1 Approaches to replication

Replication in DHTs has been approached in three primary ways. Neighbour replication stores a replica of each key value pair at each of the k closest nodes in the network. In path replication, key value pairs are replicated along the response path to queries. A final approach is multi-publication, in which data is stored at several distinct locations in the network, often by including an addition nonce into the value before hashing it.

In the DHash protocol built on top of CHORD [25], the node with the smallest id

larger than the key is deemed "responsible" for that key. The responsible node then ensures that replicas of the data are maintained at each of the $k-1$ closest nodes in its successor list. Because node ids are randomly distributed throughout the network, these k replicas are not likely to be geographically close together, and therefore are unlikely to simultaneously fail.

I elected to use a similar mechanism that adapted to the S-CHORD topology and routing protocol. Keys are stored at the the responsible node and the $(k-1)/2$ closest nodes in both the clockwise and counterclockwise directions. Thus when a node fails the key is still immediately available at the closest remaining node, which is newly responsible for it.

3.6.2 Maintenance Protocols

In order to ensure that keys are stored at the appropriate nodes, two maintenance protocols are used. A global replication protocol ensures that data that is out of place in the network proceeds towards the correct location. This occurs in a periodic "vacuuming" check performed by each peer. The peer determines for each stored key whether its id is within the k closest node ids. The keys and data which it should not store are then sent to either its predecessor or successor in the network, whichever is closer to the desired location of the key. This serves to combat any "drift" of keys in the network away from their intended location.

The local protocol ensures that all the nodes which should maintain replicas of a key indeed have it. Such a mechanism can either be eager, replicating keys as soon as a change in the network is detected, or lazy/periodic. Several authors have noted [15, 40, 42] that eager protocols can have a positive feedback effect in which a transient failure causes replication that overloads the network, causing more failures and yet greater load. In order to avoid this eventuality I chose to use a periodic, gossip based replication scheme based on the one implemented in Rollerchain [40]. Every second, a node selects a random peer to which it is connected and reconciles any differences in state. This ensures that connected peers will eventually reach a consistent view of their shared state.

In particular, each node in a pair of nodes has a set of resources that it believes should be stored at both nodes. This set can be determined by considering for each key that a node stores, whether both are within the k closest currently in its routing table.

3.6.3 Considerations for resource discovery

The replication schemes implemented in DHash and other distributed hash tables are designed for synchronising a relatively small number of relatively large objects. Thus it is quite efficient to exchange the entire list of newly stored keys when two peers synchronise.

For example, the peers might be storing 1MB chunks of a file, each associated with a 20 byte sha160 key. However in this system, we expect the size of many objects stored at the nodes to be quite small. Depending on what metadata is stored with each resource, this could be as little as just over twice the size of the key. Thus it would be particularly inefficient to send a complete list of all stored hashes at every synchronisation.

3.6.4 Set Reconciliation

Synchronising nodes' local state is therefore a problem of efficient set reconciliation. Several approaches have been described and implemented in different P2P systems, but I elected to implement one based on a Bloom filters.

A Bloom filter is a space-efficient probabilistic data-structure that allows for testing set membership. It operates by using a set k hashing algorithms, $h_1, \ldots, h_k$ and an initially zero bit vector a of length m. To add an object x to the set, we set the bit $a[h_i(x)]$ for each i. When testing membership for an object y, we check whether $a[h_i(x)]$ is zero for any i. If this is true, then y is definitely not in the set. If no such bit is zero, then y is probably in the set. For a particular number of items, we may choose the parameters of the Bloom filter, m and k, to set a particular false positive rate.

Thus for a given expected false positive rate, the space required by a Bloom filter is still linear in the number of elements to be inserted. It is, however, very efficient. For example, for an expected false positive rate of 1% we require only 9.6 bits per element.

Because of this, if each exchange sent a Bloom filter of the all the shared keys, then each synchronisation exchange would still have an $O(n)$ communication cost in the number of stored elements. However since the number of peers that must be synchronised with is small (as it is only the node's neighbours in the network), we can maintain a record of the time of the last synchronisation exchange between peers. When sending a synchronisation message, we include in the filter all relevant resources stored since the last exchange. Upon receiving a message, a peer checks whether the filter contains each resource that it has stored since it last synchronised with the originating node. It then sends any missing keys back to the originator of the synchronisation. Note that this only requires locally consistent time, not time that is consistent between the peers.

In order to further improve the efficiency of the scheme, nodes also store the last filter they received from each of their neighbours. If a node is randomly selected for which the shared state has not changed locally, a synchronisation message is sent that requests a comparison with the last received Bloom filter. This reduces synchronisation traffic if the write load is low.

As described, the scheme is probabilistic. Because there is some probability of a false positive for each inserted element, nodes may fail to recognise missing resources. There are methods for exact set reconciliation at additional computational cost [26, 22], but

during testing I found that expected key lifetimes were already much longer than an appropriate republication period. The probability due to key loss because of omission in several synchronisation exchanges can also be made arbitrarily small by setting the parameters of the Bloom filter appropriately. For the tests described in the evaluation (section 4.3) I used an expected false positive probability of 1%.

3.7 Resource Store

This class handled the storage of resources. It allowed the peer to retrieve local matches to a query as well as removing all of the data associated with a keyword for which it was no longer responsible, and optionally persisted the data to disk using Chrome's persistent storage API.

3.8 Peer

The peer object then handles any RPCs appropriately, and executes 4 periodic actions: checking its neighbours for new elements for its predecessor or successor list, fixing its fingertable entries, establishing new connections with other peers, or synchronising data with a neighbour.

3.9 Security

Decentralised systems can be vulnerable to a number of attacks. If we consider an attacker whose goal is to execute a denial of service attack (DoS) which prevents the storage and retrieval of resources, or to falsify resources, then Baumgart et al. [12] gives an overview of common attacks that would be applicable. Attacks include, but are not limited to:

- Sybil Attacks - if there is no limit on the number of node ids that an attacker can obtain, then they can damage the network in a number of ways. For example, they can artificially increase churn to cause denial of service.
- Eclipse attacks - in an eclipse attack, the attacker generates peers so that all messages to some portion of the overlay network are routed through at least one malicious node. Thus they can eclipse that portion of the network, e.g. prevent any data from being routed to it or falsify responses from that section.

These attacks can greatly reduce the performance for the user, and in the currently described system could be executed at very little cost.

The security courses I have taken have emphasised that developing and implementing a correct security protocol is "non-trivial". Thus all the security protocols considered for this project were drawn from carefully reviewed and well cited descriptions of a security protocols for DHTs, primarily those outlined for S/Kademlia in [12].

This protocol is composed of 3 key parts:

- Digital signatures for message validation
- Secure PeerId assignment
- Multiple disjoint path routing

3.9.1 Digital signatures for message validation

If each peer generates a public/private keypair, then this can be used to digitally sign the messages that it sends. This ensures that an attacker cannot spoof messages from other peers in the network, which would allow them to "steal" ids from other nodes. (For example, to perpetrate a DoS using the ids of other nodes seen in the network). This has overheads, as will be discussed in section 3.9.4.

3.9.2 Secure id assignment

Secure peer id assignment is designed to increase the cost of acquiring an id in the network. Combined with digital signatures for message validation, this can be used to greatly increase the cost of an attack. For example, the effectiveness of storage DoS can be limited by only accepting up to a maximum number of store requests from one id. Thus an attacker would have to acquire numerous ids to carry out a damaging DoS.

There are two popular mechanisms of implementing secure peer id assignment - crypto-puzzles, that require some proof of work to validate an id, and supervised signatures [12].

Crypto-puzzle schemes bind id generation to tasks that are computationally hard to solve but quick to verify. Thus peers can include this "proof of work" in their messages and quickly check the validity of the ids included in incoming messages. As a result, honest users can easily pay the cost of generating and validating ids, but an attacker seeking to generate many ids would have to pay a much larger computational cost.

Unfortunately, because of the extreme heterogeneity of peers in many systems, crypto-puzzles can be difficult to balance. If the difficulty level (e.g. the amount of computation required to discover a valid id) is too low, then they pose no barrier to an attack. If the difficulty level is too great, then legitimate peers will struggle to join the network.

The "supervised signature" scheme uses a combination of digital signatures and a central signing authority. This authority signs the public keys of peers that it trusts (along with a time interval after which re-certification is required), and can determine which peers to trust based on any of a number of centralised mechanisms, for example a CAPTCHA. This hinders the rate at which ids can be acquired by an attacker.

3.9.3 Disjoint Path Routing

In order to prevent eclipse attacks, the implementers of S/Kademlia use multiple disjoint lookup paths for each key. This reduces the probability of attack success for a given number of randomly distributed malicious nodes.

3.9.4 Implementation difficulties

Despite the fact that these security concerns lay outside the scope of the original project proposal, I had hoped to implement the security scheme described above. However after having implemented id and message validation using the open source implementation of the Ed25519 digital signature algorithms provided by TweetNaCl [13], I ran into performance issues. Whilst the validation rate offered by the implementation was sufficient to run a single peer under testing load, validating around 23 messages per second, this additional computational made it impossible to run larger networks on my development machine[6]. Testing of the security scheme could also have been done independently of the rest of the system; but I was also concerned that producing a convincing evaluation of the security properties of the system in addition to the evaluation of the routing, search and replication subsystems would represent an infeasible amount of work. During my reading on the security of P2P systems I also discovered that the mainline DHT[7] operates without any of the security features outlined above. I therefore elected to try and produce a well tested project that relies on the honesty of the users, and relegate further discussion of security to future work.

[6] I also found benchmarks for the other available JS cryptography libraries that report similar performance

[7] The largest public DHT, used as a distributed tracker by numerous torrent clients

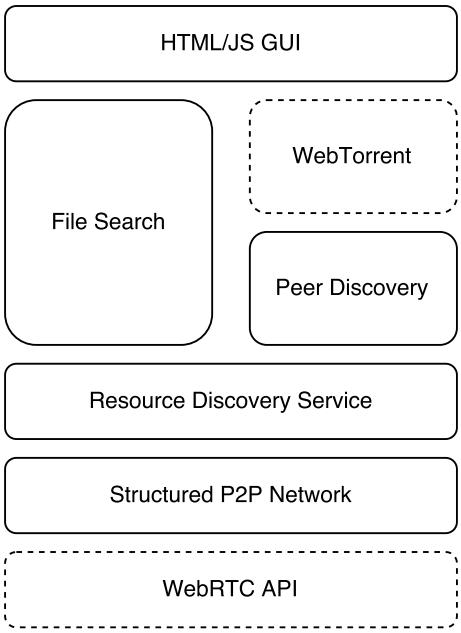


Figure 3.3: An architecture diagram for the system. Higher level components rely on the APIs offered by boxes below them. Dotted boxes indicate the pre-existing components that were not developed as part of the project.

3.10 Example Site

The project was then used to implement "trackerless" torrents and provide an example search application that demonstrates the system's resilience. Users "bootstrap" the site by navigating to an appropriate URL. The page is then automatically cached locally using the browser's "Application Cache", and all further communication is P2P. Thus once a peer has bootstrapped into the network, a user can rejoin by navigating to the URL of the page even if the server that originally provided the site is no longer online.

Using a small GUI written in HTML/Javascript, users can upload arbitrary files and associate them with keywords. They can then retrieve file resource ids by searching the distributed index using the GUI. Files can then be downloaded using WebTorrent, which uses the same distributed resource discovery system to find and connect to other peers that are interested in the file.

Chapter 4

Evaluation

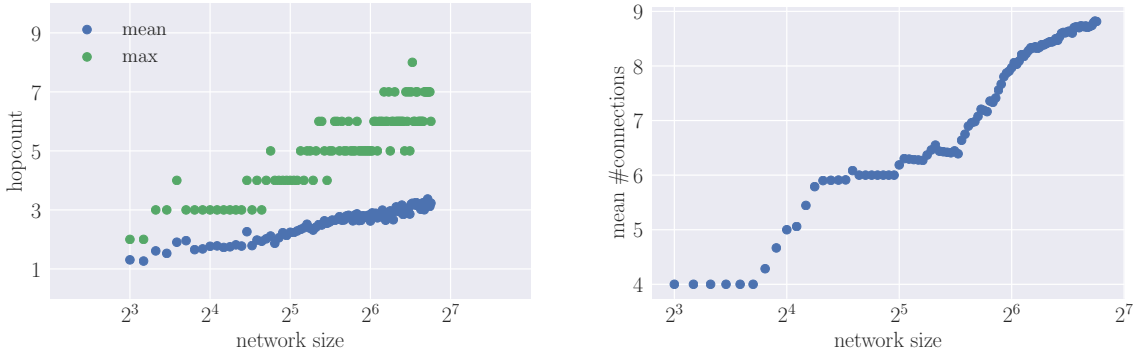
Test runs and traffic shaping tools are used to demonstrate the project fulfills all of its requirements. A comparison is then made to alternative approaches to implementing search and resource discovery.

4.1 Evaluating routing

One of the non-functional requirements of the project was:

- The system should accommodate large numbers of nodes by ensuring that local state and path lengths grow sub-linearly with the number of peers in the network.

The goal of using S-CHORD was to satisfy this requirement by ensuring that the total memory usage, the number of active connections and the number of hops between peers all grew logarithmically in the size of the network.



(a) Path lengths against network size

(b) Mean number of connections per peer against network size

Figure 4.1: State and path length scale logarithmically.

For these tests, no latency or bandwidth restrictions were introduced. Peers were brought up one by one on the local machine. After a peer was started, it performed the bootstrap routine and connected to appropriate neighbours and fingers. It then pings every other node in the network[1]. The number of hops in the outgoing and return message paths were recorded, as was the number of connections maintained by the peer. Figure 4.1 shows the results of these tests, that demonstrated that both number of connections and path lengths do indeed scale logarithmically in the size of the network.

The number of connections is not the only state stored locally by the Peer. Throughout the development of the project I strived to ensure that state was only stored for those peers included in the node's routing table, as this was designed to be logarithmic in the size of the network. To ensure that no programmer error had resulted in state leaking linearly, another test was run. A peer was set up and had its `connectTo` and other methods mocked out. It was then sent messages from randomly generated peer ids for 12 hours, simulating membership of a network of around 16 million nodes, and the memory usage monitored. The memory usage of the peer did not grow linearly (or indeed increase meaningfully at all) throughout the test, strongly suggesting that no state was being stored for every observed node. Thus this non-functional requirement was satisfied.

4.2 Evaluating search

Making a realistic evaluation of a distributed system can be complex. Several methods of testing were considered - ideally, real users could have been recruited from a variety of geographic locations to test the system from their personal machines. This is impractical for more than a small number of peers.

Another alternative evaluation scheme would have been to simulate the system using one of a number of popular network emulators [27, 11]. However this would require implementing any missing features from the emulator (as none currently include the full WebRTC stack), and a parallel implementation of much of the project in a different language. Any disparities between either the simulated stack and the real world, or between the real and simulated implementations could invalidate the results of such a test.

Instead I chose to evaluate exactly the code that would run on the end user's machine. This was done using a virtual network between containerised processes on a single host, with artificially introduced bandwidth and latency limitations.

More specifically, the system was tested as follows. A python script initialised one Docker container for each of the peers to be run. Once the peer has loaded, it contacts a bootstrapping server to fetch the id and host of a globally addressable peer. It also

[1] A central server was used to record all nodes that had been brought up and instructed each new peer to ping each one

contacts a control server to download a list of actions to perform, including a set of resources to upload; a set of queries to perform and a lifetime. The peer then connects and performs its actions over the allocated lifetime before leaving the network. A time-stamped log of all messages sent and received by each peer is saved to disk, and these can then be combined after the test is completed to evaluate performance.

The unix tools "netem" and "tc" were used to shape the bandwidth and latency characteristics of each container. In order to determine appropriate values for these control variables, I used representative figures from large measurement study on the Gnutella P2P network [46]. Because Gnutella (at the time of the study) did not use any proximity neighbor selection or similar latency-reducing schemes, the median values from this study are appropriate to use as characteristic of average links in a P2P network. Use of these values does have some limitations - network characteristics may have changed substantially since 2003, and there is no guarantee that the users of the system will have similar links to those on the Gnutella network. However, the median values taken for bandwidth and latency were 1000Kbps and 100ms respectively. Demonstrating that the system can operate on these comparatively slow links should provide good evidence for its real-world feasibility.

A 20 minute test run was repeated 6 times at each level of churn. All error bars show the 95% confidence interval calculated from the results at each level.

The performance of the system was measured using search terms that were derived from the TREC WT10g dataset. This derived dataset was produced for the evaluation of a hybrid content based p2p search system developed at Carnegie Mellon. [36, 37]. The dataset contains queries that reflect the real-world distribution of search terms in a retrieval system. In order to test the network, a centralised control server was used to distribute queries and resource identifiers to upload amongst the test peers.

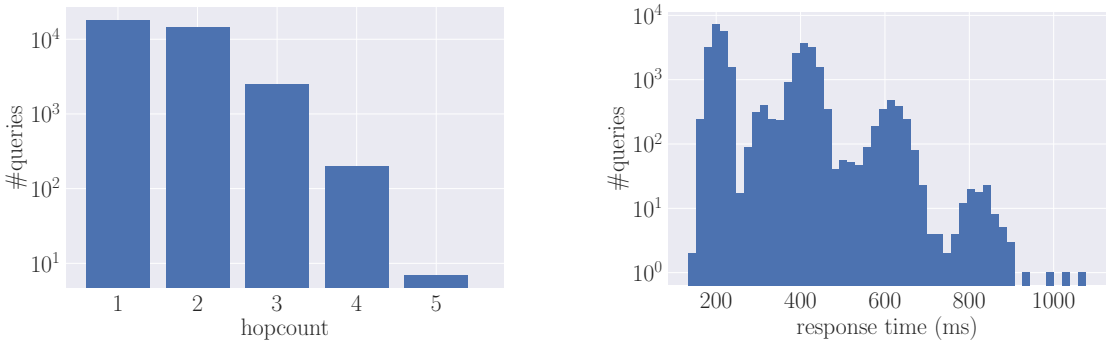
There are hard limits on how much churn such a network can handle. All other barriers aside, at some point the network will reach bandwidth saturation, after which the number of stored resources is so large that they cannot be transferred to a new peer within its lifetime. These tests were performed with many fewer resources than this, in order to verify that the basic implementation of the replication scheme was working, and that realistic latencies could be handled.

4.2.1 Stable Networks

Three requirements related to search. They noted that the system must:

1. Allow peers to store resources associated with a set of keywords.

2. Allow peers to specify a lifetime for stored resources, up to some maximum. Thus stale resources do not persist in the network indefinitely. Resources may have their



(a) Query path lengths

(b) Response times

Figure 4.2: Characteristics of a stable network

lifetime extended by republication.

3. Support conjunctive queries to discover resources. More specifically, a user or client application should be able to execute a search by providing a set of keywords. If a resource identifier stored in the distributed index is associated with any superset of that query set, it should be returned by the network. If there is no such resource identifier, then an empty result set should be returned.

Requirement 2 was tested by running a small network and uploading resources with a short lifetime. They were quickly deleted from all nodes, satisfying the requirement.

In order to determine whether criteria 1 and 3 were met the virtual network of 32 peers was run for 20 minutes. After a brief stabilisation period of one minute[2], nodes began to upload resources identifiers with keyword tags from the CMU dataset and search for resources. The correctness of the system was determined by offline evaluation after this run. A canonical inverted index was built by scanning the log file and inserting all resource identifiers uploaded by the peers. I then compared the results returned by the network's distributed index to the results of the centralised index. This allows us to determine whether the correct results were returned for each query.

The participating peers each uploaded approximately one resource id every 10 seconds and performed one search every second (some jitter was added to decorrelate network activity that introduced some variation in the upload and query rate of each peer). In the stable network, 100% of the $\sim 30,000$ queries were responded to correctly in every test run, fulfilling functional requirements 1 and 3.

Figure 4.2 shows the distribution of path lengths and query latencies for all queries that required an RPC. This again shows that the routing scheme is effective in ensuring that all queries are serviced in a logarithmic number of hops.

[2]The first minute represents very fast network growth in percentage terms, and is therefore not representive of normal operation. Thus it was discarded. This is the approach taken in [40, 48, 38]

4.2.2 Performance under churn

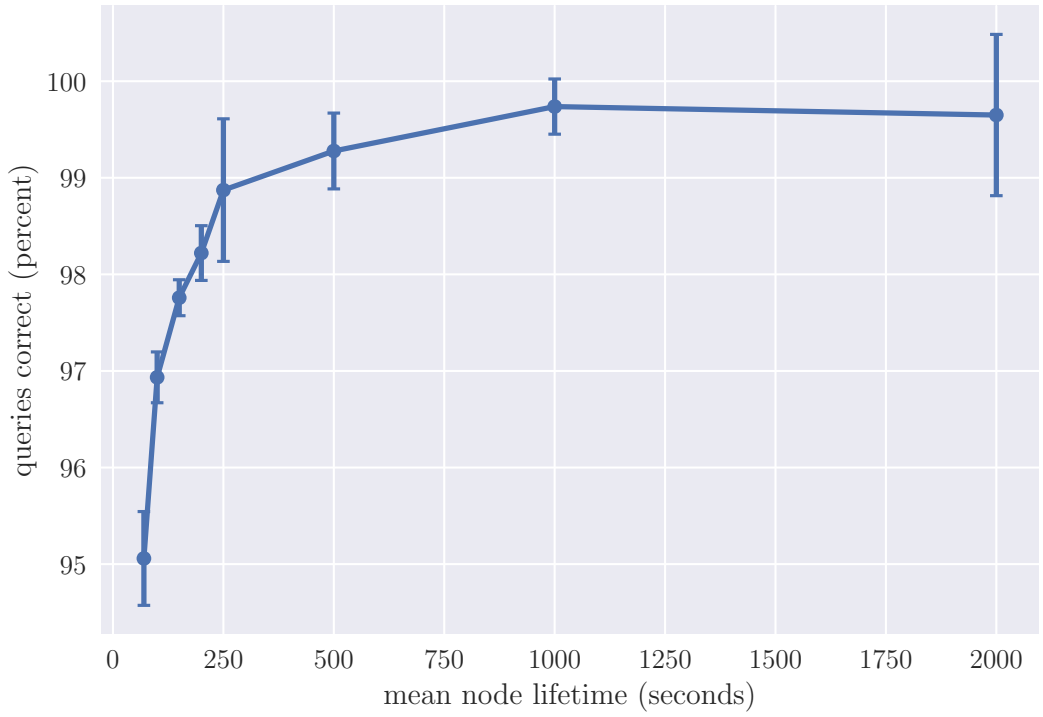


Figure 4.3: As the lifetime of the nodes increases, more queries are responded to correctly.

In any realistic deployment of the network, nodes will join and leave unexpectedly. It is important that the system continues to function under such conditions, but we should expect some performance degradation.

Peers were given a Weibull distributed lifetime to mirror the distribution of web-page dwell times found in [34] and several measurement studies of large p2p networks [49]. This allows us to determine how the replication mechanism will respond to churn in a reasonable scenario.

The mean page dwell time found in [34] was just above 70 seconds. This was used as a lower bound on peer lifetime, as it seems unlikely that peers using the service would spend less time in the peer to peer network than they would on a random web page.

Greater levels of churn increase the replication traffic and RTC signalling traffic, because more connections must be negotiated and more new nodes require replicas. This is illustrated by figure 4.7. However, even with very short node lifetimes, which are equal to the average dwell time on a web page as measured by [34], the overhead of maintaining the overlay network is manageable on the average links suggested in [46].

As churn increases, a small percentage of queries are also responded to incorrectly as shown in 4.3. Responses were marked incorrect if the set of resources returned by the

distributed index was not exactly the set of resource ids that had been uploaded. Even at the highest expected churn, however, more than 95% of queries were serviced correctly.

4.3 Evaluating Replication

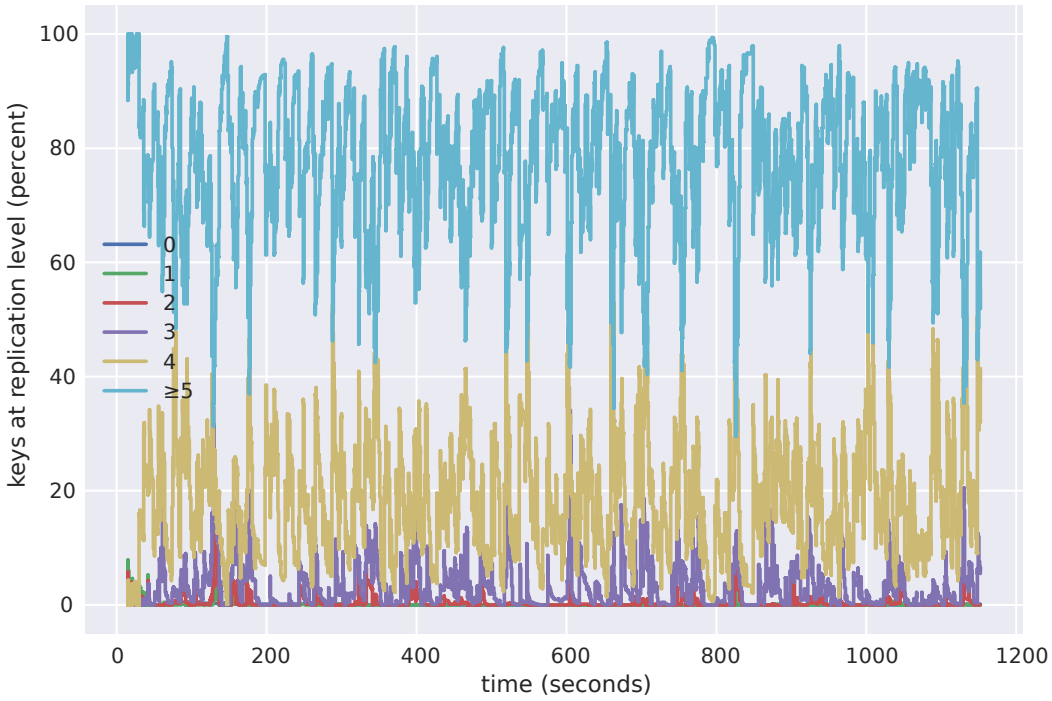


Figure 4.4: Most of the resources in the network stay at or above the desired replication level, even at high levels of churn.

One of the non-functional requirements of the project was that the system should:

- Ensure that the average lifetime of stored resource identifiers can exceed the average uptime of a peer in the network.

In order to achieve this, keys in the system are replicated. I refer to the number of nodes storing a particular key as that key's replication level. For these tests, the target replication level was set to 5. Figure 4.5 shows that the mean replication level is close to the intended value even as nodes join and leave the network rapidly. Figure 4.4 shows the proportion of keys at each replication level throughout a 20 minute simulation. Note that despite the frequent loss of keys due to nodes leaving the network that the system quickly re-replicates keys and ensures that most keys stay at or above the desired replication level. These results show that replication system functions as intended.

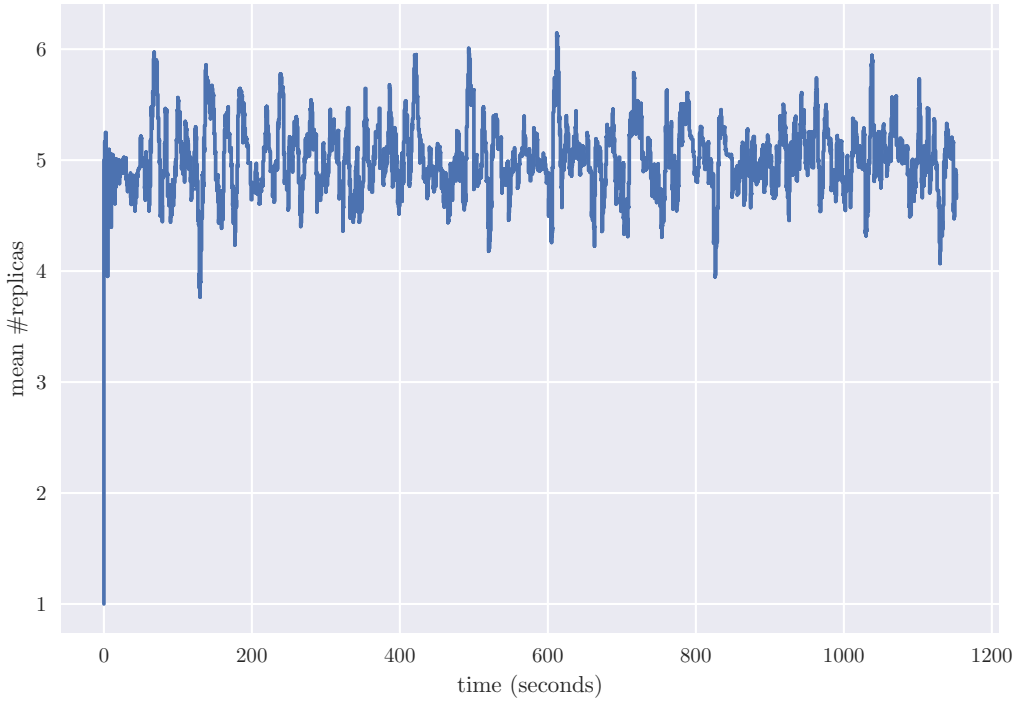
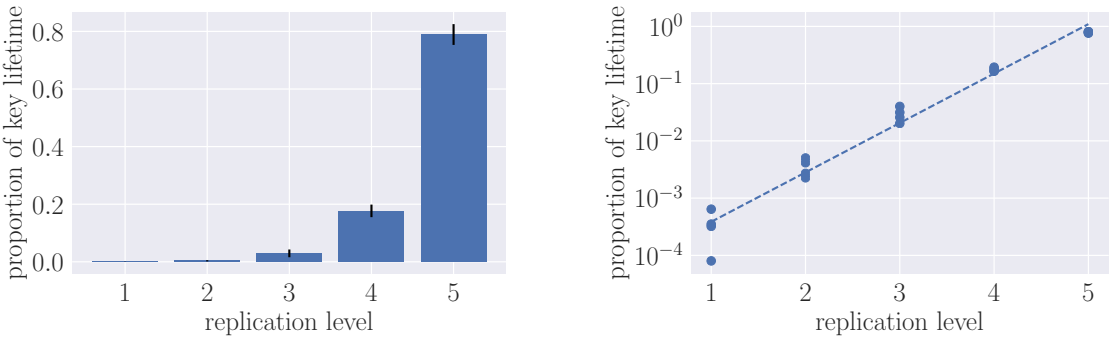


Figure 4.5: This plot shows that the mean number of replicas per key stays close to the preset value of five in a network of 32 nodes with a mean lifetime of 70 seconds.



(a) Proportion of key lifetime at each level of replication.

(b) A linear model against the logarithm provides a good fit.

Figure 4.6: The proportion of the key lifetime spent at each replication is exponentially distributed.

A key is lost from the system when its replication level is 1 and the node storing it fails. We can plot the proportion of time that a key spends at a particular level of replication after its initial uploader sees the success response, and note that it appears exponentially distributed (figure 4.6a). Fitting a linear model against the log of the data using the Python package "SciPy" [31] also supports this hypothesis, with a p-value well below 0.001.

We may then approximate how long resources will be stored by the system at different levels of churn. If the mean time between node failure is l then a resource is lost if it is stored on a single node and that node fails. Given that the key is stored on only a single node for some known proportion of its lifetime p_1. Because a peer's randomly selected id and the peer's lifetime are independent, we can note that the expected key lifetime is then given by l/p_1.

Using the results from the 6 tests with a mean node lifetime of 70s, I determined that $p_1 = 0.0004$. This gives a mean time between key loss of around 50 hours with the simulated network conditions. Thus when the network operates without bandwidth saturation, the replication mechanism works well and ensures that keys persist for much longer than the nodes hosting the service. Further, the exponential relationship demonstrates that increasing the replication level is an effective means of increasing key lifetime. Thus this non-functional requirement was satisfied.

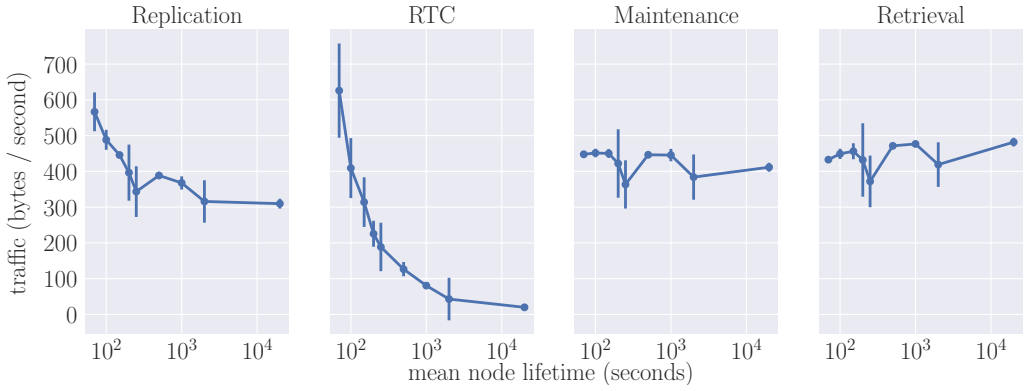
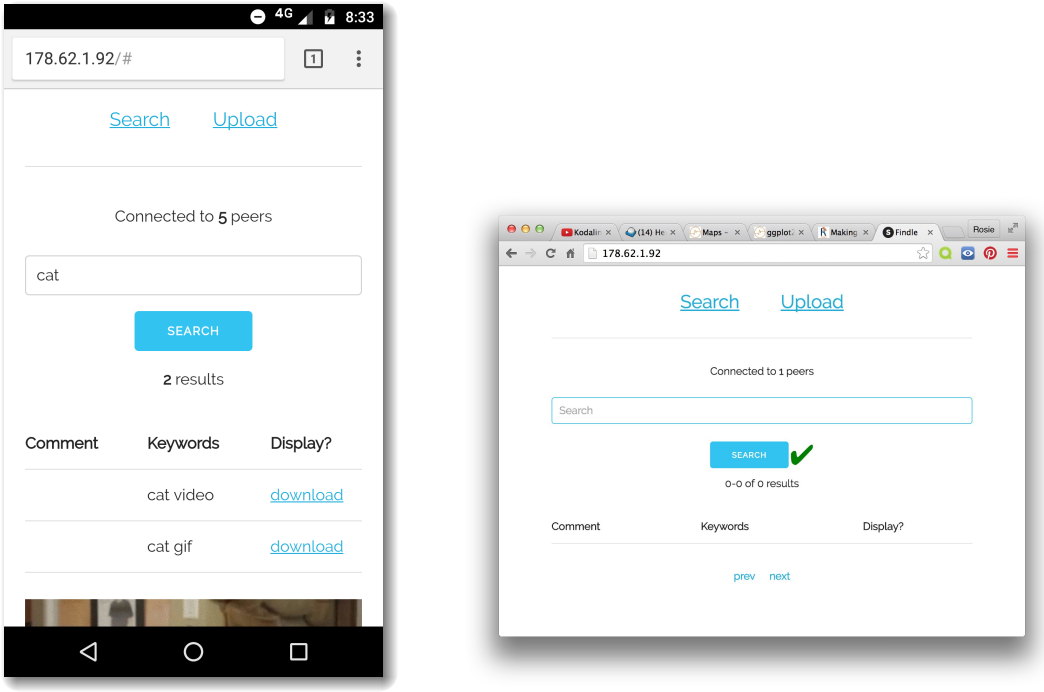


Figure 4.7: Traffic volumes decrease as the node lifetime increases

4.4 Example site

Having evaluated the resource discovery mechanism, the example site was tested manually. This was sufficient to indicate that the interface between the search page, the bittorrent implementation and the underlying resource discovery implementation was correct. A bootstrap server initially hosted the site. A publicly addressable peer, and several browsers then loaded the page. Several files were uploaded, searched for and downloaded. The web server that hosted the bootstrap page was then disabled, and it was verified that the peers would automatically load the page from the application cache and re-connect to



(a) Android phone (b) Mac computer

Figure 4.8: The example site works on mobile Chrome and over (my) cellular network.

the networ. Thus they could still search for, upload and download files. The final non-functional requirement, that the system should:

- Allow peers that have a cached copy of the code to join the network even if all the developer's machines have failed.

was therefore satisfied.

Screenshots of the example site are available in figure 4.8.

4.5 Comparison to alternative approaches

We can compare this approach to alternative approaches to resource discovery. Firstly, it makes sense to consider how this mechanism compares to distributing the complete index to every peer. In the following analysis

k = replication level
w = average number of words associated with a resource
l_{kw} = average length of a keyword in bytes
n_{peers} = number of peers in the network
n_{res} = number of resources stored

We then calculate the average amount of data stored at each peer by the system. Note that an index entry including the resource identifier and the complete keyword list is approximately wl_{kw} bytes. This must be stored at w responsible peers in the network, and is replicated $k-1$ times around each responsible peer. Thus the average amount of data from the index stored for each peer is given by:

$$\frac{w^2 l_{\text{kw}} n_{\text{res}} k}{n_{\text{peers}}}$$

In a system in which every peer stores the complete index, each node must store at least $n_{\text{res}} w l_{\text{kw}}$ bytes. Thus the project's mechanism, of storing part of a larger index with redundancy, is more efficient in terms of total storage when $n_{\text{peers}} > wk$.

The total number of transferred bytes for uploading a single resource in the "total replication" model, assuming optimistically that in that system a peer can route a message to every other in 1 hop, is given by $n_{\text{peers}} w l_{\text{kw}}$. By contrast this mechanism must route a message of size wl_{kw} to each of the w responsible nodes, who then each replicate the resource to $k-1$ close peers. Thus the total number of bytes transfered is $w^2 l_{\text{kw}}(k + \log n_{\text{peers}})$ in the system produced here.

Thus this mechanism is much more efficient than total replication if the number of peers is large.

We should also compare the distributed resource discovery scheme to a centralised method. The following table outlines some key differences, assuming that the developer runs a single peer.

	Centralised Index	P2P Index
upload time	$O(1)$	$O(\log n_{\text{peers}})$
query time	$O(1)$	$O(\log n_{\text{peers}})$
storage cost for developer	increases with more users	decreases with more users
bandwidth cost for developer	increases with more users	decreases with more users
cost of increased availability	developer responsible for every replica	more peers increase availability for free
system complexity	low	high

Thus we can see that this mechanism is not always appropriate. If a developer needs to operate a small, simple system without many users, and may bear responsibility for hosting any or all replicas of the data, then a centralised scheme will offer users reduced query latency. However, for developers that need to eschew the hosting costs of a highly available search system used by a large number of peers, this project offers a possible solution.

Chapter 5

Conclusion

This project implements a P2P system that can be joined simply by navigating to a web page once. Evaluation under pessimistic network conditions shows that the system continues to function well under churn, and an example application shows it can be used for distributed file sharing. By implementing key based routing and a generic resource discovery service, the system provides three features frequently requested by developers in the open source community, allowing for decentralised WebRTC signalling [7], trackerless torrents in the browser [8] and decentralised search [7]. These tools could provide the foundation for future systems that are easier to access and more resilient to failure than ever before.

5.1 Future work

There are many exciting avenues for future work, both in terms of improving the core project and using it to create new decentralised applications for the browser. Improvements could include completing and expanding the security protocol or introducing more complex mechanisms for ranking results.

By building on top of the resource discovery service or the key based routing system, developers could implement one of any number of protocols that rely on either of these primitives. This could include efficient decentralised publish subscribe systems such as SCRIBE [21], or decentralised streaming systems such as DeMSI [51]. The authors of the latter note that "resource discovery is one of the most critical components of this infrastructure".

Because KBR also offers an efficient broadcast mechanism, the routing layer of this project could be also be for broadcast based P2P system in the browser. Systems that rely on broadcast include most cryptocurrencies, as well as systems like Structella that support a search protocol with arbitrarily complex queries [20].

Each of these systems could be built top of this project and optimised for the browser, making them easily accessible to many more users.

5.2 Reflection

Producing the system was challenging. Working in two new languages and learning the continuation passing style (which was required to handle the asynchronicity of a P2P system in the single-threaded environment of the browser) was non-trivial. Modifying classic approaches such as CHORD to ensure that they could accommodate the need for negotiating each new connection without relying on a central server also took careful thought and experimentation.

The most enjoyable aspect of the project was familiarising myself with the wealth of research that has been done into P2P technology; but it was also fun to draw on so much of the material from my undergraduate courses. Deciphering the existing technologies and selecting an appropriate one to fulfill the project's goals needed an understanding of networking, distributed systems and computer systems modelling, and actually implementing the system then drew upon numerous other courses.

Bringing these various threads together to transform this previously theoretical understanding into a practical system was then immensely rewarding.

Bibliography

[1] Bintrees: binary trees for javascript. https://github.com/vadimg/js_bintrees. Accessed: 2016-12-22.

[2] Napster homepage (from november 27, 1999). "https://web.archive.org/web/19991127080306/napster.com". Accessed: 2017-01-20.

[3] Stun ip address requests for webrtc, github. https://github.com/diafygi/webrtc-ips. Accessed: 2017-01-22.

[4] Texcount web service (version 3.0.0.24). http://app.uio.no/ifi/texcount/online.php. Accessed: 2017-05-11.

[5] Typescript: Javascript that scales. https://www.typescriptlang.org. Accessed: 2017-04-17.

[6] Webtorrent. https://webtorrent.io. Accessed: 2016-10-12.

[7] Webtorrent github issue #186. "https://github.com/webtorrent/webtorrent/issues/186". Accessed: 2017-01-22.

[8] Webtorrent github issue #288. https://github.com/feross/webtorrent/issues/288. Accessed: 2017-01-22.

[9] Jeannie Albrecht, David Oppenheimer, Amin Vahdat, and David A Patterson. Design and implementation trade-offs for wide-area resource discovery. *ACM Transactions on Internet Technology (TOIT)*, 8(4):18, 2008.

[10] Salman A Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017*, 2004.

[11] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79–84. IEEE, 2007.

[12] Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure key-based routing. In *Parallel and Distributed Systems, 2007 International Conference on*, pages 1–8. IEEE, 2007.

[13] Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. Tweetnacl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America*, pages 64–83. Springer, 2014.

[14] Ashwin R Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. In *ACM SIGCOMM computer communication review*, volume 34, pages 353–366. ACM, 2004.

[15] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[16] Eric A Brewer. Towards robust distributed systems. *PODC*, 7, 2000.

[17] Jake D Brutlag, Hilary Hutchinson, and Maria Stone. User preference and search engine latency. 2008.

[18] J. Buford, H. Yu, and E.K. Lua. *P2P Networking and Applications*. The Morgan Kaufmann series in networking. Elsevier Science, 2009.

[19] Matthew Caesar, Miguel Castro, Edmund B Nightingale, Greg O'Shea, and Antony Rowstron. Virtual ring routing: network routing inspired by dhts. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 351–362. ACM, 2006.

[20] Miguel Castro, Manuel Costa, and Antony Rowstron. Should we build gnutella on a structured overlay? *ACM SIGCOMM Computer Communication Review*, 34(1):131–136, 2004.

[21] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[22] Josh Cates. *Robust and efficient data management for a distributed hash table*. PhD thesis, Massachusetts Institute of Technology, 2003.

[23] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418. ACM, 2003.

[24] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[25] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 202–215. ACM, 2001.

[26] David Eppstein, Michael T Goodrich, Frank Uyeda, and George Varghese. What's the difference?: efficient set reconciliation without prior context. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 218–229. ACM, 2011.

[27] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell, and J Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14, 2008.

[28] Ian Hickson, Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, Anant Narayanan, and Bernard Aboba. Webrtc 1.0: Real-time communication between browsers. Working draft, W3C, September 2016. https://www.w3.org/TR/2016/WD-webrtc-20160913.

[29] Ryan Huebsch, Joseph M Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 321–332. VLDB Endowment, 2003.

[30] Ryan Huebsch, Joseph M Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with PIER. *Proceedings of the 29th international conference on Very large data bases - Volume 29*, pages 321–332, 2003.

[31] Eric Jones, Travis Oliphant, and Pearu Peterson. {SciPy}: open source scientific tools for {Python}. 2014.

[32] Yuh-Jzer Joung, Li-Wei Yang, and Chien-Tse Fang. Keyword search in dht-based peer-to-peer networks. *IEEE Journal on Selected Areas in Communications*, 25(1):46–61, 2007.

[33] Gunnar Kreitz and Fredrik Niemelä. Spotify - Large scale, low latency, P2P music-on-demand streaming. *2010 IEEE 10th International Conference on Peer-to-Peer Computing, P2P 2010 - Proceedings*, 2010.

[34] Chao Liu, Ryen W White, and Susan Dumais. Understanding web browsing behaviors through weibull analysis of dwell time. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 379–386. ACM, 2010.

[35] Lintao Liu, Kyung Dong Ryu, and Kang-Won Lee. Keyword fusion to support efficient keyword-based search in peer-to-peer file sharing. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 269–276. IEEE, 2004.

[36] Jie Lu and Jamie Callan. Content-based retrieval in hybrid peer-to-peer networks. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 199–206. ACM, 2003.

[37] Jie Lu and Jamie Callan. Peer-to-peer testbed definitions: trecwt10g-2500-bysource-v1 and trecwt10g-query-bydoc-v1. `http://boston.lti.cs.cmu.edu/callan/Data/`, 2003. [Online; accessed 03-Jan-2017].

[38] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[39] Valentin A Mesaros, Bruno Carton, and Peter Van Roy. S-chord: Using symmetry to improve lookup efficiency in chord. In *2003 International Conference on Parallel and Dis-tributed Processing Techniques and Applications (PDPTA'03*. Citeseer, 2002.

[40] Joao Paiva, Joao Leitao, and Luis Rodrigues. Rollerchain: A dht for efficient replication. In *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 17–24. IEEE, 2013.

[41] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 21–40. Springer-Verlag New York, Inc., 2003.

[42] Sean Rhea, Dennis Geels, Timothy Roscoe, John Kubiatowicz, et al. Handling churn in a dht. In *Proceedings of the USENIX Annual Technical Conference*, volume 6, pages 127–140. Boston, MA, USA, 2004.

[43] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 99–100. IEEE, 2001.

[44] Jonathan Rosenberg. Interactive connectivity establishment (ice): A methodology for network address translator (nat) traversal for offer/answer protocols. 2010.

[45] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[46] Stefan Saroiu, P Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. *Proceedings of Multimedia Computing and Networking*, 2002:1–15, 2002.

[47] Charles Severance. Javascript: Designing a language in 10 days. *Computer*, 45(2):7–8, 2012.

[48] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[49] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202. ACM, 2006.

[50] Dan Wing, Philip Matthews, Rohan Mahy, and Jonathan Rosenberg. Session traversal utilities for nat (stun). 2008.

[51] Alan Kin Wah Yim and Rajkumar Buyya. Decentralized media streaming infrastructure (demsi): An adaptive and high-performance peer-to-peer content delivery network. *Journal of Systems Architecture*, 52(12):737–772, 2006.

Appendix A

Dependencies & Build System

Javascript did not originally have a module system. Because of the scope of this project, I used the Gulp build system along with Browserify to automate transpiling Typescript to Javascript and bundling the JS into a single file for delivery to the browser.

The project also had some dependencies, including the following:

```
// The AVA unit-testing framework.
ava@0.16.0
  author: 'Sindre Sorhus <sindresorhus@gmail.com> (sindresorhus.com)',
  url: 'git+https://github.com/avajs/ava.git'

// Balanced binary trees.
bintrees@1.0.1
  author: 'Vadim Graboys <dimva13@gmail.com>',
  url: 'git://github.com/vadimg/js_bintrees.git'

// Fast bloom filters.
bloomfilter@0.0.16
  author: 'Jason Davies (http://www.jasondavies.com/)',
  url: 'http://github.com/jasondavies/bloomfilter.js.git'

// Bundle files for the browser.
browserify@13.3.0
  author: 'James Halliday <mail@substack.net> (http://substack.net)',
  url: 'git+ssh://git@github.com/substack/node-browserify.git'

// A build system that automates transpilation.
gulp@3.9.1
  author: 'Fractal <contact@wearefractal.com> (http://wearefractal.com/)',
  url: 'git+https://github.com/gulpjs/gulp.git'
```

```
// A binary serialization framework.
msgpack-lite@0.1.26
  url: 'git+https://github.com/kawanet/msgpack-lite.git'
  author: '@kawanet'
```

Appendix B

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Browser-hosted P2P Resource Discovery

N. McAleese-Park, Gonville & Caius College

Originator: N. McAleese-Park

10 October 2016

Project Supervisor: Dr J. Singh

Director of Studies: Dr G. Titmus

Project Overseers: Dr D. J. Greaves & Prof J. Daugman OBE

Introduction

We increasingly interact with devices through network connections. The rise of the "Internet of Things" means that speakers, lights and numerous other "things", both in our home and elsewhere can now be controlled through our phones, laptops and other devices.

This new trend poses some challenges. Interaction with many current-generation things requires communication with a central server, which has both privacy and availability issues. Specifically, all the interaction with a local device is mediated, managed and logged by the central system, which also represents a single point of failure. It would

be a frustrating and potentially dangerous user experience to lose control of the local light manager because of a distant server problem.

Why are many of these interactions centralised? Outside of the economic motivations of the device providers, there are three primary practical reasons.

1. Users do not wish to install specialist software to interact with every device they encounter. A centralised server can offer a web interface that allows a user to interact with the device through the browser.

2. A centralised and globally addressable server can allow the user to connect to devices that are not globally addressable, by acting as a gateway.

3. Users need to discover applications, services and devices. It is impractical for a user to uncover and recall a specific identifier for each thing – instead they want to interact with a service that will show them all the devices associated with, for example, an account or location.

However a resource discovery service does not have to be centralised. A decentralised, peer-to-peer, network can have a number of advantages over a centralised system. If well-designed they can use built-in redundancy to offer substantial uptime, as well as reducing costs by distributing load over the end users' machines. In fact, as the number of connected devices increases, it may become infeasible to manage all resource discovery centrally. Large peer-to-peer networks can also help alleviate privacy issues by ensuring that no single party holds an invasive amount of data about any one user, including that user's actions in the network.

Access to most current-generation peer-to-peer networks requires the installation of specialised software, which often only serves a single purpose (such as file sharing). Recently however, browser to browser communication been standardised, in the form of "WebRTC", a specification for real time browser to browser communication motivated by the demand for in-browser voice and video calls [28]. As a result, peer-to-peer systems can be hosted on browser based clients. This means that any client with a browser environment could join the network, including users on desktop as well as in mobile apps and browsers.

This offers the exciting opportunity to use WebRTC to deliver the advantages of decentralised resource discovery whilst facilitating easy, browser-based access to the system for potential new users.

The project

The aim of this project is to develop a peer-to-peer distributed search application that allows a client to efficiently discover sets of resources.

In the general case, this could be by the specification of an arbitrary set of constraints on the resource meta-data. However this work will focus on a *keyword search* interface for resource discovery.

More specifically, we assume a large set of tuples (K_i, r_i) each of which associates a set of keywords with a resource identifier. The user then presents a query set of keywords Q, we aim to return all r_i such that $Q \subseteq K_i$

Keyword search was chosen as the method of resource discovery was chosen because it offers a substantial degree of flexibility, and has been implemented in P2P systems with some success [41, 32]. Research has been done into more expressive languages for P2P resource discovery, including SWORD, but these systems have struggled to overcome performance issues and are much more complex [9].

A user or application might be searching for files, things, or arbitrary other components, such as services. Example queries that the system might be used for include:

1. cambridge computer lab light manager room 5
2. ubuntu linux 64bit
3. cat videos
4. 0x961ec519d2d818f2b238f

Use in such a variety of situations presents some challenges.

In a system designed for generic resource discovery, we need to be able to find both popular and rare (potentially including private) resources. This prevents the use of an completely unstructured P2P system, as whilst these systems can be effective for finding popular resources, they either use flooding, which is unscalable, or random walks which have either less than 100% recall or unbounded latencies [30].

The project will therefore involve implementing a structured overlay network. Whilst this is a well studied problem in general, there are no widely deployed browser based implementations. Deployment in this new environment will present some challenges.

One problem which will be need to be overcome is "signalling". The WebRTC specification assumes that there is an external communication channel available in order to initially establish communication between two peers.

This raises two problems: firstly, the classical bootstrapping problem of a P2P network is worsened. We do not need simply the address of a peer in the network, but instead an active communication channel with one. Further, a signalling channel will be needed for every new peer connection. If a separate server is used, this will add some centralisation to the network. However once the overlay network is set up, the P2P network itself may be used to transmit data between nodes.

During the initial work on the system, a bootstrapping server will be developed to allow new peers to connect to the overlay. Whilst this does provide a single point of failure for the current model, it is worth noting that a dedicated bootstrapping server is not strictly necessary: the required handshake could be done over any channel, such as an instant messaging service, or local bluetooth communications. Alternatives to this point of centralisation will be considered as future work.

Starting point

The design of structured P2P overlay networks has been well studied, and possible systems include Chord, Kademlia or Pastry [38, 48, 45]. These systems were initially designed for use as distributed hash tables, but the routing mechanisms they employ can also be adapted to P2P keyword search [41, 32].

As for the implementation of P2P networks with WebRTC, the most successful project is WebTorrent, which replicates the bittorrent protocol over WebRTC connections. The project does not, however, currently implement the bittorrent DHT, and offers no search or resource discovery functionality.

I have some hobby experience writing both JavaScript and languages which transpile to it (estimated 1000 lines of code).

I have taken several relevant undergraduate courses, including Networking, Software Engineering and Concurrent and Distributed Systems.

Work to be done

Completing the system will be broken down into several sub-problems:

1. Implement a structured overlay network on top of WebRTC. This will involve implementing an addressing scheme for peers in the overlay and a routing mechanism that ensures peers can efficiently forward messages between each other.
2. Research and implement a redundancy system that ensures that loss of resource identifiers from the distributed index requires multiple simultaneous node failure.
3. Use these layers to implement the keyword search functionality by distributing an inverted index over the overlay network.
4. Determine what query and index data will be used to test the system. The demand for resources indexed by the system will not be uniform. Determining a suitable statistical model, or acquiring a real world set of resource queries will ensure that the tests of the system are under realistic loads.

5. Test the system in an environment without node churn, ensuring that the system meets the success criteria specified below.

6. Evaluate the effect of node churn on the performance of keyword search. Vary the rate of node failure, and record the how this increases the latency of responses. Determine what level of node churn is required before resource identifiers begin to be lost from the distributed index. This is particularly important in a browser hosted environment, as closing a tab is sufficient to remove a node from the network.

Success criteria

For the project to be considered a success, several conditions must be met. They will be evaluated in a simulation network run on a single machine.

1. The developed software should execute in a modern web browser. Specifically, Chromium v54 will be used for testing the following success criteria.

2. A user or client application should be able to execute a search by providing a set of keywords. If a resource identifier stored in the distributed index is associated with any superset of that query set, it should be returned by the network. If there is no such resource identifier, then an empty result set should be returned.

3. Resource identifiers should only be lost from the distributed index when multiple nodes fail faster than the network can recover. More specifically, there should should be a replication parameter k, and a recovery procedure. Resource identifiers should only be lost if more than k nodes fail before the recovery procedure can complete.

The system will then undergo further evaluation. This will include running the network under increasing rates of node failure and recording the latency of query responses, as well as the amount of traffic generated by the recovery mechanisms. Once the failure rate becomes high enough, it is expected that the network to fail to retrieve some resource identifiers, and this failure rate will also be recorded against the rate of node failure. The results of these experiments could then be compared to performance studies of widely deployed p2p systems to determine the real-world viability of the project. One of the proposed extensions includes running the system in volunteer browsers, and if this work is completed then the data will also be used in the final project evaluation.

Possible extensions

There are several possible extensions to this work:

1. Active resources, such as services or IoT devices, may wish to update their resource identifiers (eg because they change location, or their available capacity changes). This could be achieved by using cryptographic signatures and a timestamp to establish ownership of a particular resource identifier in the distributed index. If a client receives a number of signed identifiers, the most recent one could be selected.
2. An example user-level application or applications, such as video stream, light management, or sensor access could be developed.
3. For large result sets, clients will have preferences about how results are ordered. They may not care about all the results in the system. The protocol could be extended to allow for sorting based on resource meta-data. Examples include preferring results signed by a particular public key, or sorting by distance from a location.
4. Real peers could be solicited to run nodes. This would allow the system to be tested with real-world NAT configurations, which could provide additional evidence for its viability. This would be subject to the appropriate ethical review for requesting users to run "instrumented software", as logs of system performance would need to be recorded.

Resources required

For this project I intend to to use my own quad core Dell XPS 13 that runs Linux Mint. Backup will be both by Dropbox, for continuous backup of small changes, and to my personal bitbucket account using git. I have another similar laptop which can be used in the event of an unexpected failure of my main machine.

For the extension experiment (subject to the appropriate ethical review), friends may be recruited to run test nodes on their local machines. This will not require them to install any specialist software, but will require a suitable warning to inform users that their open browser page may consume both network and CPU resources. It should be made clear that if they consider either of these costs too large, they will be at liberty to close the browser tab and therefore leave the network.

Timetable

Planned starting date is 24/10/2011.

1. **By 4th November:**
   - Learn the finer details of the WebRTC API.
   - Further literature review of structured overlay networks and resource discovery approaches.

- Select an appropriate overlay network.
- Implement a small centralised signalling server.
- Connect two browsers using WebRTC.

2. **By 18th November:**
   - Set up a testing system for the network.
   - Ensure debugging logs from each node are easily accessible and that bringing up nodes is easy.
   - Begin implementation of selected routing system.
3. **By 2nd December:**
   - Complete implementation of the overlay network.
   - Implement a redundancy scheme that prevents data loss on node failure.
   - If work is to schedule, implement in-network signalling.
4. **During Michaelmas vacation, by 20th January:**
   - Implement keyword search using the overlay network.
   - Acquire a real world search data set, including queries, or determine how to appropriately model one.
   - Simulate the network under the acquired data, experiment with varying levels of churn.
   - Write progress report.
5. **By 3rd February:**
   - Extension: Solicit real peers to run experimental nodes.
   - Extension: Set up logging of real peer performance.
6. **By 17rd February:**
   - Begin writing dissertation.
   - Extension: Complete the extension by producing analysis of real peer performance.
7. **By 3rd March:**
   - Finish introduction and begin the preparation and implementation section of the dissertation.
8. **By 17 March:**
   - Finish the preparation and implementation sections of the dissertation.
9. **During Easter vacation:**
   - Complete writing dissertation.
   - Submit to supervisor by **31st March**.
   - Final revisions and dissertation prepared for submission by **28 April**.