A Report submitted in partial fulfilment
of the regulations governing the award of
the Degree of
BSc (Honours)
Ethical Hacking for Computer Security
at the University of Northumbria at Newcastle

Project Report

Ses#!

A vendor agnostic, browser based peer to peer file backup application developed using web standards.

2014/2015

Software Engineering Project

By Christopher Simpson
chris.j.simpson@live.co.uk

DECLARATION OF AUTHORSHIP

I declare the following:
(1) that the material contained in this dissertation
is the end result of my own work and that due
acknowledgement has been given in the bibliography and references to
ALL
sources be they printed,
electronic or personal.
(2) the Word Count of this Dissertation is 12252
(3) that unless this dissertation has been confirmed as confidential, I agree to an entire electronic copy or sections of the dissertation to being placed on the eLearning Portal (Blackboard), if deemed appropriate, to allow future students the opportunity to see examples of past dissertations. I understand that if displayed on eLearning Portal it would be made available for no longer than five years and that students would be able to
print off copies or download.
(4) I agree to my dissertation being submitted to a plagiarism detection service, where it will be stored in a database and compared against work submitted from this or any other School or from other institutions using the service. In the event of the service detecting a high degree of similarity between content within the service this will be reported back to my supervisor and second marker, who may decide to undertake further investigation that may ultimately lead to disciplinary actions, should instances of plagiarism be detected.
(5) I have read the Northumbria University/Engineering and Environment Policy Statement on Ethics in Research and Consultancy and I confirm that ethical issues have been considered, evaluated and appropriately addressed in this research.

SIGNED:

Special thanks to

David Kendall
For supervising this project. For answering no doubt the most bizarre and misconceived questions to ever have walked the office, but for listening anyway and having the patience for working to an answer.

Martin Wonders
I know nobody faster at responding to email, inspite of the volume of them!

Sony Ericsson, for pioneering the early WebRTC concept
Mozilla, Google, and Netflix and others for supporting the editors of the relevant specifications for the APIs on which this project so heavily relies.

Abstract

Seshi is an experimental standards compliant, web-browser based, Peer to Peer application which provides offline storage of user files using IndexedDB and supports peer to peer file transfers of those files between consenting peers using WebRTC. The application is a proof of concept, this report discussed the elements of its design, justifies its design approach and explains how its interlinking technologies (ICE, STUN, SCTP, TURN), work together to form a useful application. The application is written in Javascript and joins real time communication protocols (WebRTC, SCTP) with the offline storage API indexedDB to realise a Browser based application which supports the backup, storage and and sharing of arbitrary files between peers over WebRTC Datachannels. The report explains the development of each layer of the application: Persistent Storage, Signalling, NAT Traversal, and Datachannels in the context of web browser applications. Design challenges faced during the development process are documented, addressing topics such as avoiding SCTP connection terminations with WebRTC and design suggestions for when implementing similar peer to peer file transfer applications. The report offers a critique of the web as a software development platform today and an appreciation of the successes of the standards committees which make such cross platform application possible. Finally further critique of the developed application is presented, its practicality, and suitability as a browser based application concluding that browser based applications are beginning to genuinely challenge traditional vendor specific platform-targeted development projects.

Keywords: Offline Storage, WebRTC, Peer to peer, Filesharing, ICE, STUN, TURN

Why, what, how, why how

Basic idea behind application

This application will be an operating system agnostic solution for backing up data in a distributed manner freely without the need for a centralised service. The interface for this application should be the web browser as it is a commonly installed application and does not require the installation of additional programs or plugins.

If one were to visualise this program you can imagine a graphical folder on a web page over which people can drag files into which they wish to backup safely into this storage network. These files are chunked into smaller pieces, duplicated for redundancy, and distributed amongst participating users in the network where they may be reassembled in the future.

Should the user want to retrieve these files they should be able to access their files from anywhere using a web browser and not have to install any additional software. This would include any standards compliant web browser including those found on mobile phones, tablets and Internet enabled such as tvs if you wish to extend the analogy.

Why

In the websphere, ‘online’, ‘the internet’ it has become increasingly less common to find web services which simply provide an easy mechanism for peers to store & share files between each other indiscriminately of platform, identify or account signup. Often popular services, which provide such facilities, thrive and benefit greatly from the data collected from such submitted data, a behavior which is arguably not truly understood by the lay user. Users benefit too, because applications can adapt to user preferences and behaviors, yet, these services do thrive upon the notion that, “If you’re not paying for it, you become the product.” (Goodson, 2012). Afterall, data collection is attested to be useful, it is a form of business intelligence after all which supports more directed forms of advertising: “Deliver messages to users in the right moment based on what they’ve recently Tweeted or engaged with” (Business.twitter.com, 2015), and “On Facebook, you'll only pay to target your adverts to the exact people you'd like to connect with.” (Facebook for Business, 2015).

In a social context (such as social media websites) does data collection for business purposes constitute a form of social research? Facebook, for example, was criticised deeply for mixing business intelligence with social research publicly apologising for “poor communication over psychological experiments” (Gibbs, 2014) it ran across its platform. If so, then surely such

activities should be subject to the constraints of social research such as informed consent? Informed consent, “it is the formulation of a widely recognised moral obligation to respect others and take into account their interests.” (Homan, 2001 p.330).

This is perhaps a false problem, however, as providing users of a service are not just in agreement but also have a clear understanding over the product they use, then concrete consent exists between the user and the provider of that service. The latter is arguably unattainable; it is not realistic to expect users to understand every aspect of a systems functions, therefore merely “presumed consent” (Homan, 2001 p.330) presides, often gained via lengthy privacy policies which users either do not read or do not have the technical knowledge to understand. Though, in the meritocratic sense, it is challenging to criticise the behaviour of such organisations which successfully attract and serve millions of users a day[1] and goes further to issue public apologies responding to self-organising debates amongst its user base discussing the ethics of Facebooks’ actions, albeit afterthefact.

What this program is and what it does

This web application, program, allows users to store arbitrary files within their web browser by utilizing offline the offline storage mechanism IndexedDB addressing the use case that “user agents need to store large numbers of objects locally”(Mehta et al., 2015). Additionally, it allows two or more peers to send their stored files to each other in a peer to peer fashion without the need to a central server for transport using the WebRTC protocol suit which “allows two users to communicate directly, browser to browser” (Bergkvist et al., 2015).

The application therefore allows the persistent storage, query, and transfer of arbitrary data across peer connections which it can establish. At the time of writing, the application is cross compatible with both Google Chrome’s V8, and Mozilla’s spidermonkey Javascript engines. Signalling, which will be explained in depth, is currently achieved via a signaling server also written in Javascript using the NodeJs platform. This signalling server implementation may be expanded to an overlay network of signaling servers in the future, or, bypassed altogether by means of out-of-band communications between peers. Users are able to benefit from direct connectivity between chosen peers and are not limited in the size of files they store within their own web browsers, nor limited to the amount transmitted (only by the capacity of devices and network bandwidth).

Organisation of files stores in the system is achieved via the abstraction of ‘Box Ids’ which can be applied to a collection of files upon storying them into the system. Identification of a collection is therefore a matter of querying the data store for files of a given Box Id. Equally, individual files may be called out of out the database by their individual ‘File Id’ indiscriminately of whether a file is part of a collection or not. This permits, therefore, the creation of new ‘collections’ by combining existing ‘BoxIds’.

[1] (Investor.fb.com, 2014)

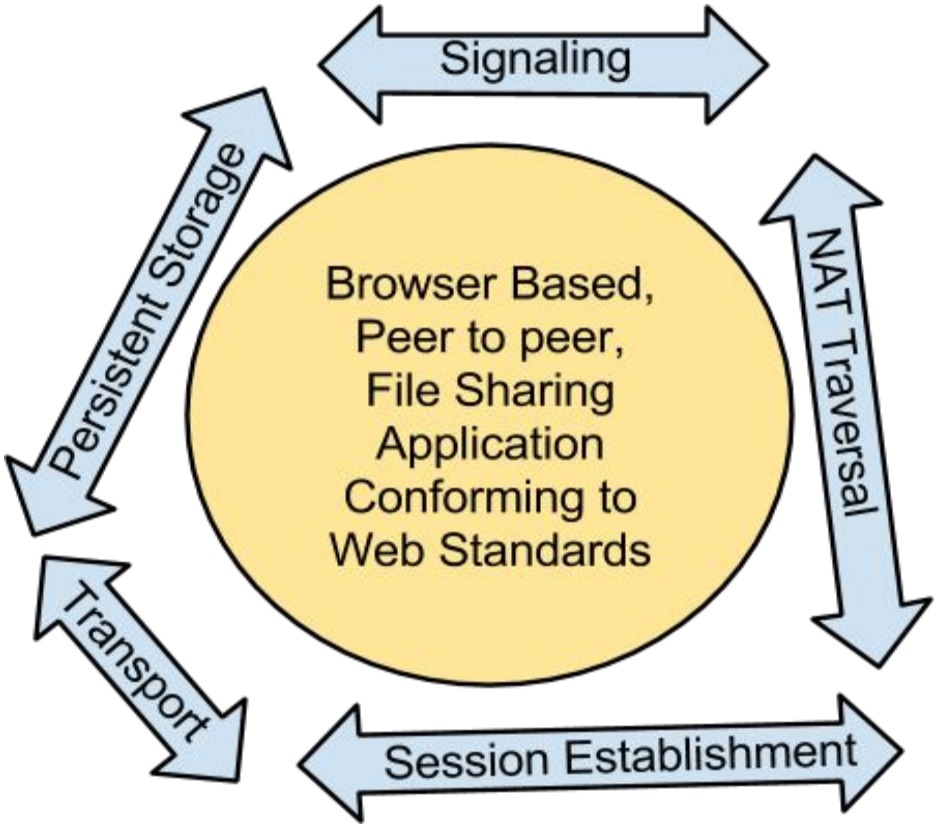
This application is essentially a non-descript object store with the ability to send data in a peer to peer fashion, on request and in real time. It may be of use to other software projects which require a reliable, interoperable, object store with addressable storage, supporting peer to peer transfer of the data stored within it. It runs on any web enabled devices which implement the WebRTC, IndexedDB, ICE, STUN, and TURN (optional) standards. Currently this includes Mozilla Firefox version 37.0.1 and Google's V8 engine which is used by Google Chrome Browser version 42, and Opera 28. Additionally, due to the application being written using web technologies, it is also compatible as a native mobile application on the Firefox OS platform, and Google's Android given they both use the same respective Javascript engines as their web browsers. Do do this, the Apache *Cordova* project is used to compile the application down into native mobile applications.

Architectural overview of the developed application

Features, decisions, problems encountered and solutions (some novel)

The important features needed to realise the program were:

1. A reliable, unlimited persistent storage mechanism which worked within a web browser.
2. Reliable session establishment between peers
3. Reliable transfer of data between those peers



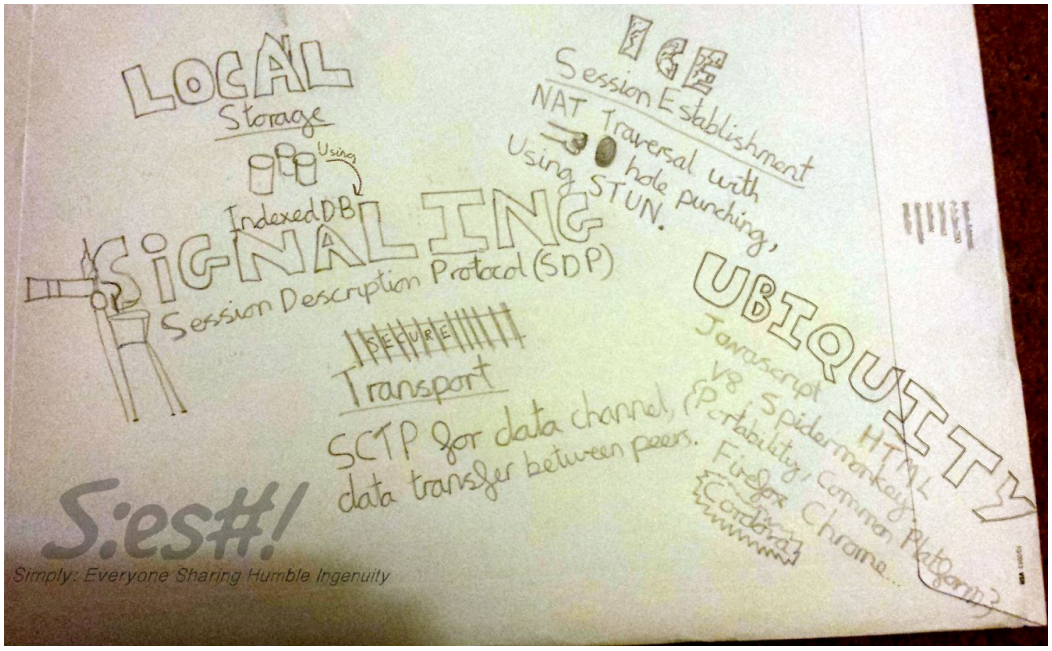
The building blocks of the application.

By folding these features into the program, a program which allows first the local backup of a user's files, and then the transmission of these files between known peers becomes possible.

It is the process of this development and encountered challenges it presented which shaped the project. The various features implemented and challenges they presented with techniques applied to work through them are now discussed.

The main technologies which came together to implement the required features are drawn below. Please refer to the higher resolution image in the appendix for closer inspection, as it provides a clear overview of the various technologies used, and how they are applied to realise the functioning product. There are many interlinking technologies used in this project which at first may seem unruly, though by looking one step back from the project it becomes easier to

see how each technology pays its role and demonstrates a clear standards-based and 'don't reinvent the wheel' approach to the applications architecture.



Drawing showing the key building blocks to realising the application: Local Storage, Session Establishment, Signaling, Transport and Ubiquity.

Although many of the stated browser technologies are indeed relatively new[2], what is especially novel and unique to this project is the careful selection and combination of browser and network technologies to realise the browser based file sharing tool, which to the best of my research has not been done before.

The developed product uses technologies only available within the browser stack to implement its requirements, as named in the picture above.

Ubiquity of the browser stack

Interoperability through the ubiquity of web technologies

In the interest of interoperability and challenge, the required features were all implemented using APIs available within the browser, which are almost solely exposed to the developer via a JavaScript interface.

Confining oneself to only the technologies available within the browser stack limits implementation options yet benefits greatly from better interoperability between operating systems and significantly lowers the barrier to entry for end users given the installation of an executable is no longer needed (providing the availability of a web browser).

[2] WebRTC was first demonstrated by Google in 2011 http://www.w3.org/2011/04/webrtc/

Simply, such self-imposed design constraints confine the available technologies to create the product to only what's available within the browser stack. Please refer to the comment 'why use the browser' for a more in depth critique of this larger design decision over developing a tradition installable executable. Essentially, this works argues that the browser has become ubiquitous, and much powerful development platform putting it in a very viable position to challenge some more traditional development practices such as development for a particular operating system.

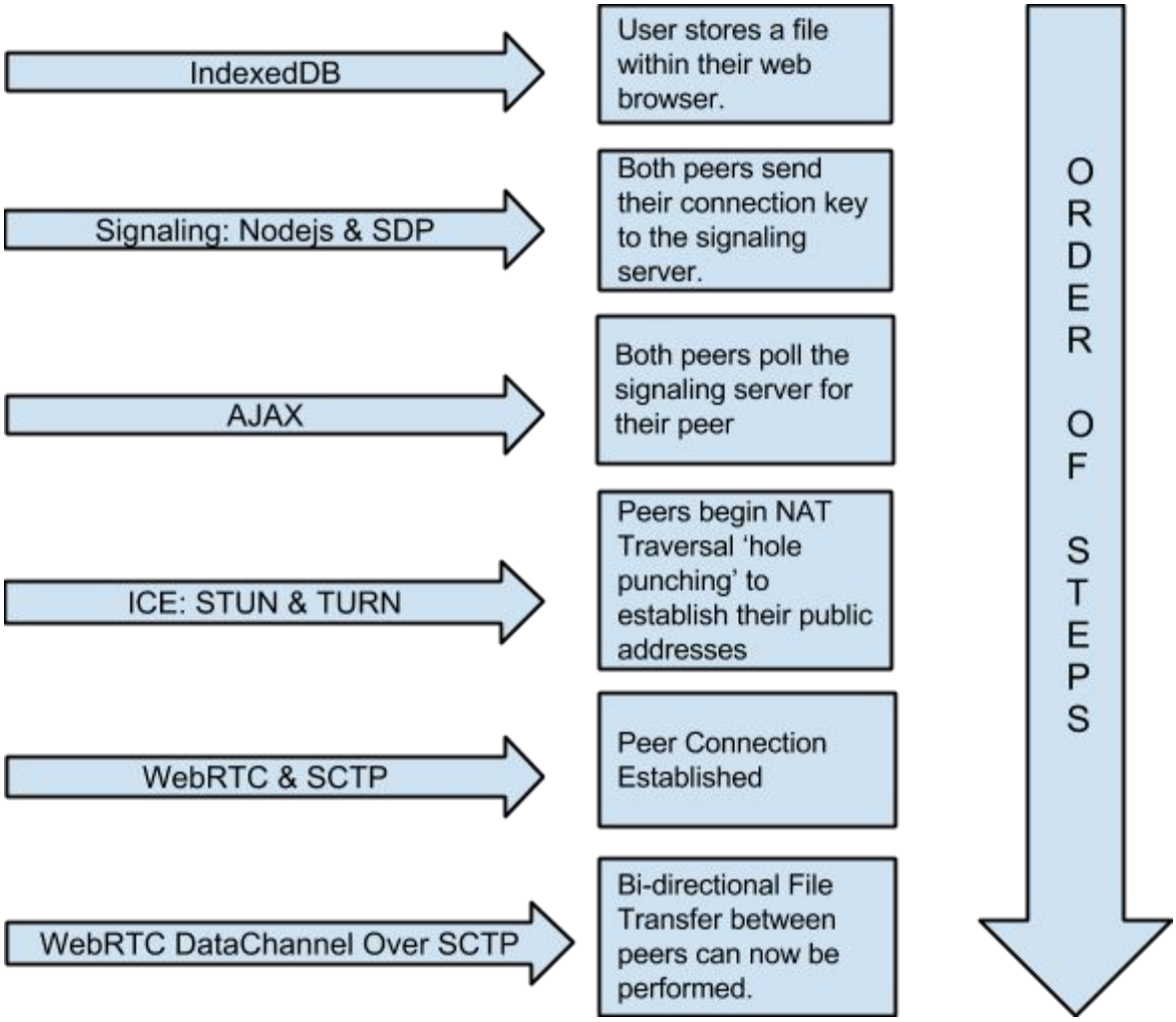
The web browser technologies, mechanisms used to realise the product included:

- Javascript
- Real time communication (WebRTC)
- File API for file manipulation
- IndexedDB for Client-side storage

Additionally, Implementations of various network protocols were used to provide assistance to the browser when performing tasks such as Signaling, and NAT traversal; both of which are needed when forming a peer connection. These included the layer three and four technologies:

- Internet Connectivity Establishment (ICE) which describes the STUN & TURN protocols for NAT traversal
- Stream Control Transmission Protocol (Transport layer)

Flow Diagram - A user's interaction with the system



Each of the stages outlined in the above diagram will now be addresses, synthesising the development process and discussing the part each layer of the application plays in making a functioning product.

Persistent Storage Mechanism: IndexedDB

Local Storage, Session Establishment, Signaling

Offline Storage within the browser Using IndexedDB

Prior to data being shared with a peer, data is stored within the web applications using 'Offline storage'. Offline storage is the ability to store data persistently client side, within the web browser.

Historically, storage of client this was limited to just 64k cookies[3] which may be set to expire, blocked or removed manually. Since then, greater capacity offline storage options are exposed to the developer. The names of the various offline storage mechanisms browsers offer today include:

- WebSQL Database
- IndexedDB
- Web Storage
- Local Storage
- AppCache

Storage Mechanism	**Maximum Offline Storage Permitted**	**Comment**
WebSQL Database	Five megabytes	Superseded by IndexedDB due to lack of browser vender interest in implementing the standard. “The specification reached an impasse” (Hickson, 2010).
Web Storage	Five megabytes, per origin	W3C Recommendation as of 30 July 2013 (Hickson, 2013)
IndexedDB	The Quota Management API defines there to be no fixed limit other than the physical disk space on the device itself. However, some vendor’s User Agent implementations do impose their own restrictions: Chrome Browser implementation: “As large as the available space on the hard drive. It has no fixed pool of storage.” (Developer.chrome.com, 2015) Firefox Browser implementation: “There isn't any limit on a single database item's size, however there is in some cases a limit on each IndexedDB database's total size. This limit (and the way the user interface will assert it) varies from one browser to another.	Persistent storage available varies depending on device, and is further limited by the Quota Management API. The “specification defines an API to query and manage usage and availability of a user's local storage.” (Yasuda, 2015)

[3] Interestingly, SCTP, the transport layer protocol used for file transfers has a 64K maximum payload for each packet.

	Firefox has no limit on the IndexedDB database's size. The user interface will just ask permission for storing blobs bigger than 50MB.” (Mozilla Developer Network, n.d.)	

Depending on the mechanism used, these features and their corresponding APIs allow a website to request to store gigabytes of data within the visiting users web browsers

To address the more substantial file storage requirements of the application, three local storage options were evaluated for use with ‘IndexedDB’ being selected as the most appropriate for the task at hand.

Chosen storage model: IndexedDB

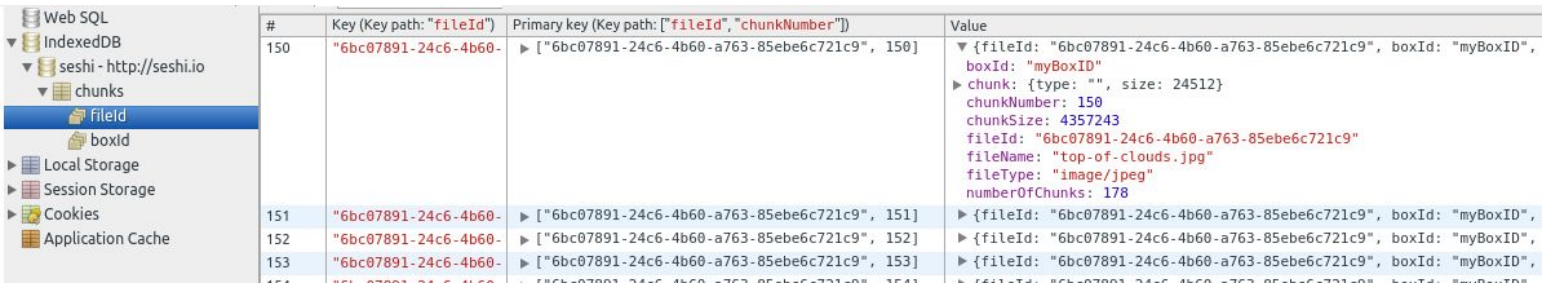
IndexedDB storage was identified as the most suitable mechanism to store user data locally within the browser. According to the W3 specification, IndexedDB is “a concrete API to perform advanced key-value data management … It does so by using transactional databases to store keys and their corresponding values (one or more per key), and providing a means of traversing keys in a deterministic order.” (Mehta et al., 2015). It is flexible in that objects of any type may be stored in the object store, is persistent storage, and fast (it uses “persistent B-tree data structures that are considered efficient for insertion and deletion as well as in-order traversal of very large numbers of data records.”).

IndexedDB requires first a Database name to be defined, followed by any object stores within the database which will must be named. Finally, keys can be placed on the objects in those stores for efficient searching, insertion and deletion.

For the applications' file storage model, the following object store was defined:

Database name	Seshi
Object Store name	chunks
Keys	fileId, boxId

Using a combination of the File API, and IndexedDBB, the application stores users files persistently in their web browser using the scheme defined as above. Note that indexedDB is purely an object store with key paths defined. This is not a relational database, the chosen indexes on the fileId and box Id are further explained below using the same JSON format in which all objects are stored.



Each object stored in the IndexedDB database is simply a 'chunk' of a user submitted file with some metadata attached. These objects (referred to as 'chunks' from now on) may have either been stored directly by the user of that User Agent, or received over a peer to peer datachannel using WebRTC and SCTP and stored into this database by issuing the relevant IndexedDB API calls.

Every chunk is stored into the object store in the following format as a JSON object. The code below shows both the properties of this and subsequently the interaction with the IndexedDB API to store chunks into the object store:

```
function addChunkToDb(blob) {
        //Add chunks to indexedDB
        db.transaction("rw", db.chunks, function() {
        var test = blob;
        console.log("Storing chunk number: " + window.curChunk.chunkNumber);
        db.chunks
        .add({
                boxId: window.boxId,
                chunk: blob,
                chunkNumber: window.curChunk.chunkNumber,
                chunkSize: window.curChunk.chunkSize,
```

```
            fileId: window.curChunk.fileId,
            fileName: window.curChunk.fileName,
            fileType: window.curChunk.fileType,
            numberOfChunks: window.curChunk.numberOfChunks
        });
        }).then(function() {
        //Transaction completed
        console.log('Time to show file back to user/recipricant..?');
        }).catch(function(error) {
        //Transaction failed
        console.log(error);
        });

}//End addChunkToDb
```

Function addChunkToDb(chunk) demonstrates IndexedDBs' support for transactions, which is useful to ensure the integrity of the stored objects. 'blob' is an instance of type Blob, which is a raw binary chunk of containing part of the complete file.

Storing files as chunks rather than single blobs.

Note, before each file is added into the IndexedDB store, it is chunked on the fly using Javascripts 'File API' slice() function. The individual chunks are inserted one at a time. "The slice method returns a new Blob object with bytes ranging from the optional start parameter upto but not including the optional end parameter" (Ranganathan and Sicking, 2015).

Storing files in this way, as opposed to storing entire files as a single object, makes the sending of files between peers more flexible and more reliable. In this way, only the chunks needed may be sent to peers (thus reducing bandwidth requirements).

Additionally, the transfer protocol used to sending files between peers (SCTP) imposes a maximum payload of 65k per packet, which again furthers the cause for chunking stored data. However, when attempting to send chunks of this size to a peer, corruption was a common issue. The discussion toward to Datachannel further explores these challenges, and solutions identified. Though they are discussed briefly here.

It was later identified that the reasoning for this file corruption was inherent in SCTP implementations: it is recommended by the Chrome WebRTC development team to reduce this payload even further to as low as 16kb:

"Messages longer than a few (*cough* 16 *cough*) kb are unlikely to work well. SCTP doesn't have large buffers internally for storing these while transmitting/fragmenting/defragmenting/etc. Please break them up into chunks when sending, and de-chunkify on the other side. It'll be

much nicer to the memory pressure in chrome tab, and the system as a whole. It'll also scale quite well to much larger sizes." (Code.google.com, 2013)

In taking this advice, data corruption was eliminated allowing the reliable transfer of chunks between peers. Lastly, using smaller objects per insert also gets around some browser vendor restrictions impose in their implementations of offline storage which limit each stored object size to 50mb (as Firefox does). Not, however, collectively there is not a limit on how many objects are stored, only the size of each individual object.

The establishment of connections between peers is discussed next.

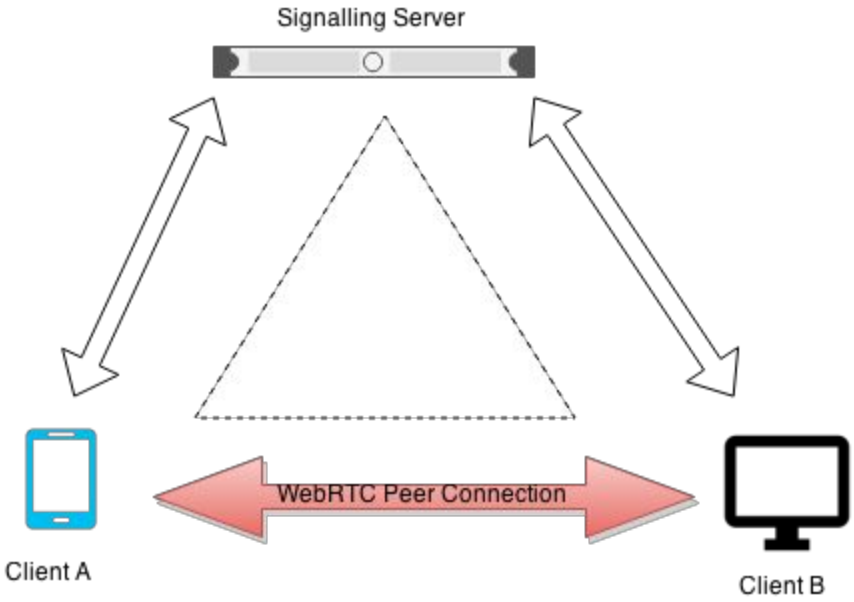
Signalling

NAT traversal

Network address translation is topic of larger debate covering much wider internet connectivity issues and is not limited to real time web communications. However, it is impossible to avoid, defining and discussing the NAT traversal issue given the extent to which it affects internet facing peer to peer applications such as the one developed during this project.

What is signaling, why is it needed, and how was it implemented & why how

> "Signaling plays an important role in WebRTC but is not standardized, allowing the developer to choose. This lack of standardization and multiple options has resulted in some confusion. A number of different signaling approaches have been proposed and used, and an understanding of the differences between the approaches is useful in selecting the right one for a given WebRTC application."
> - Johnston and Burnett, 2014



Almost all elements of the process for establishing peer-to-peer connectivity between browsers is signposted from the WebRTC standard which discusses:

- Connecting to remote peers using NAT-traversal technologies such as ICE, STUN, and TURN.
- Sending the locally-produced streams to remote peers and receiving streams from remote peers.
- Sending arbitrary data directly to remote peers.

As alluded to, however, the developer is left in the dark when it comes to Signalling. This leaves questions unanswered by the WebRTC specification such as: How do peers discover each other? As well as: how do peers indicate that they wish to communicate with each other at all?

Therefore, the selection, design and implementation of the Signalling server was, as Johnston states a required consideration given “an understanding of the differences between the approaches is useful in selecting the right one for a given WebRTC application.” (Johnston and Burnett, 2014). What follows therefore is the reasoning and justification for the design of the Signaling facet of this project’s application.

Why is signaling not standardised?

Signalling, through required for peer to peer connectivity to happen, does not need to be prescribed by an W3 recommendation “because it does not need to be standardized to enable interoperability between browsers” (Johnston and Burnett, 2014 p.70) . That is, any form of communication may be used for signalling.

As stated, any method of communication can potentially be used to perform the signaling function. The purpose of signaling is simply to allow two or more parties to make it known where they are, and how they might be contacted. To be clear, it is not even required for this process to be electronic. There is no constraint over the means by which a signaling message reaches a peer. Though less practical, messages may also be passed via out-of-band communications, for example on paper, or by using spoken word. “Communications are coordinated via a signaling channel which is provided by unspecified means, but generally by a script in the page via the server, e.g. using XMLHttpRequest [XMLHttpRequest]” (Bergkvist et al., 2015). As per the W3 comment, the XMLHttpRequest object was used as the mechanism to facilitate signaling.

Cynically, out of band communications such as those described (spoken word, paper) are slow, impractical and too complex for the average user. For this reason the compromise of developing a signaling server was completed which removes the need to manually exchange session information, and instead the use of an agreed ‘key’ between two peers was adopted as a mechanism to marry two peers initially. The key is a text string chosen by the peers to identify their communication channel used entirely just during the signaling process, as the peer to peer connection establishment is standardised collectively by the WebRTC, ICE, STUN and TURN protocols which are discussed as much as necessary to explain the development of this application. Their purpose can be briefly summarised as:

“When two peers decide they are going to set up a connection to each other, they both go through these steps. The STUN/TURN server configuration describes a server they can use to get things like their public IP address or to set up NAT traversal. They also have to send data for the signaling channel to each other using the same out-of-band mechanism they used to establish that they were going to communicate in the first place.” (WebRTC, Bergkvist et al., 2015)

The risk of developing a signaling server

Finally, asking peers to pass keys via a signalling server as a solution to the non-standardised signaling process is a weak point of this system, and indeed to the understanding of WebRTC as a whole. Again, “This lack of standardization and multiple options has resulted in some confusion” (Johnston and Burnett, 2014). It is a source of confusion given developing a signalling server risks giving wind to an assumption that the application and WebRTC is not peer to peer. It is easy to mistakenly presume that all communications go via the signaling server for example, yet this would be grossly mistaken, as a concrete example: Once the WebRTC process is complete and a peer connection established, the signalling server may be switched off with the peer connection left totally unaffected.

Such lack of standardisation over signaling signifies the danger in developing a signaling server, which is compromise to make the signalling process easy for the end user. It creates the illusion of needing to use a certain proprietary signalling server implementation in order to partake in using the application. This is not the case, it simply makes becoming a peer easier, and is more practical than peers exchanging signaling information manually.

The development of a signaling server

A very simple signaling server was developed for this application. It is a simple message passing server using the NodeJS platform. NodeJs server applications are written in Javascript (“Node.js,” 2015), which simplifies development given the common language between server and client applications.

Signaling server implementation

Importantly, the choice of using Node.js as the platform for developing a signaling server is its suitability given the single-threaded, non blocking, model it adopts making it preferable to the real-time demands of the Signaling server. It additionally eliminates the need to concern oneself with message passing between processes on multi-threaded implementations such as Apache and reduces the wastefulness of busy waiting and the kludge ("locking problems, memory challenges”(Dahl, 2009)) caused when working against a multi threaded models striving for high concurrency. The original author of Nodejs claims this is the bane of most web application servers “Threaded concurrency is a leaky abstraction. In many cases, [servers are] just waiting for the response.” (Dahl, 2009).

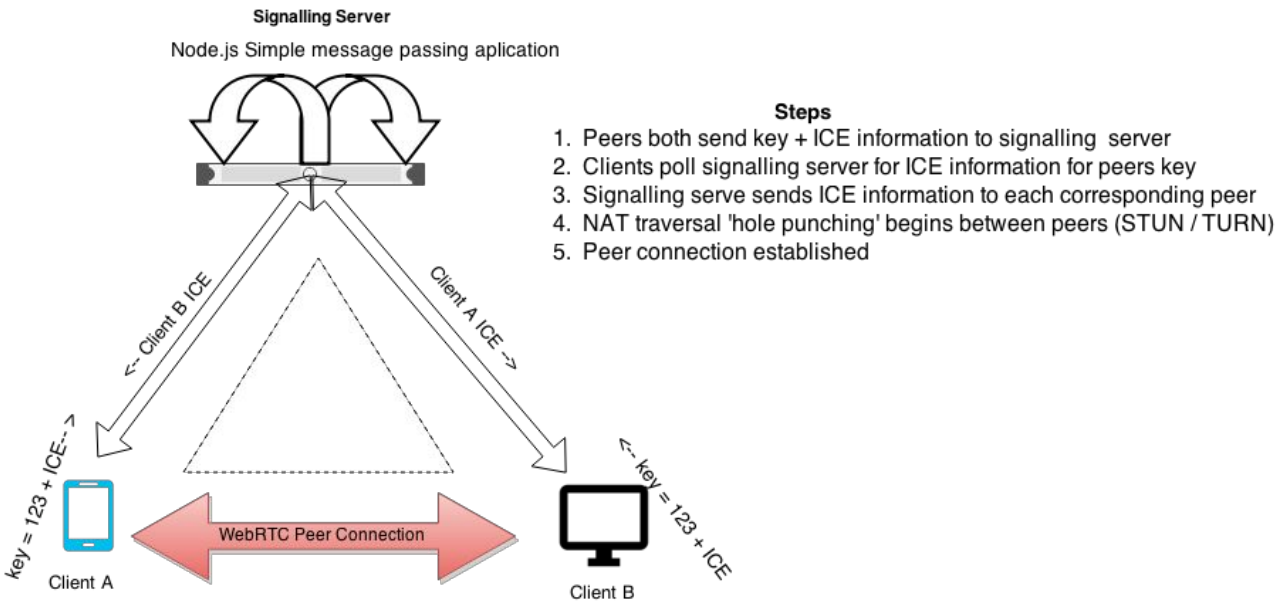
The prior justification for adopting NodeJs simply made development easier in that respect, as it also went hand in hand with the real-time nature of the peer to peer application. The signalling server needed to be fast, ideally non blocking, and needed only to serve the very simple purpose of message passing between two peers as fast as possible. “Context switching between the different threads, which is what Apache does, is not free, is costs some cpu time” (Dahl, 2009).

Developing the Signalling server on on multithreaded web server such as Apache, or similar would have introduced interesting design challenges (such as message passing between threads, as well as meeting the desire for a near real time signaling server). However, it was deemed an unnecessary complication given the availability of node.js as a platform and in that “it dramatically simplifies the communication between separate browser requests” (Johnston and Burnett, 2014 p.61).

The contents of signaling messages

What messages do signaling server(s) contain, what is their function?

The diagram below expands upon the previous diagram adding further clarification to the role that the signaling server plays in the peer to peer application. Peers both send a key, which is the same between both peers, and their own ‘ICE candidates’ this information is used by the peers to attempt a peer connection between each other . Interactive Connectivity Establishment (ICE) and its role in combination with the developed signaling server is now explained.



ICE Candidates

ICE candidates carry a text string containing one or more 'Candidate transport addresses', which usually amount to an IP address and port number.

ICE Protocol
"A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols" (RFC5245)

RFC5245 explains these may include:

- A transport address on a directly attached network interface
- A translated transport address on the public side of a NAT (a "server reflexive" address)
- A transport address allocated from a TURN server (a "relayed address").

The notion of a 'candidate' addresses rather than simply a single 'peer address' is due to the fact each client cannot know unequivocally, ahead of establishing a connection whether a connecting peer can infact connect directly to it. This is due to many internet clients residing behind NATs, Firewalls and or other potential network related obstructions to establishing true peer to peer connectivity. Addressing this challenge is the essence of the ICE protocol. It requires each peer to form a collection of *potential* address which a peer may be able to reach it by. Once collected, peers send their ICE candidates to a signaling server for an expectant peer to collect in the hope that at least one of the addresses is indeed usable as a transport address to connect to the peer.

ICE Candidate extract from application

The extract below shows a typical ICE candidate message, which is generated by calls to the relevant WebRTC client API call (RTCPeerConnection.onicecandidate) which alerts the application to the fact transport candidate addresses have now been identified. These ICE

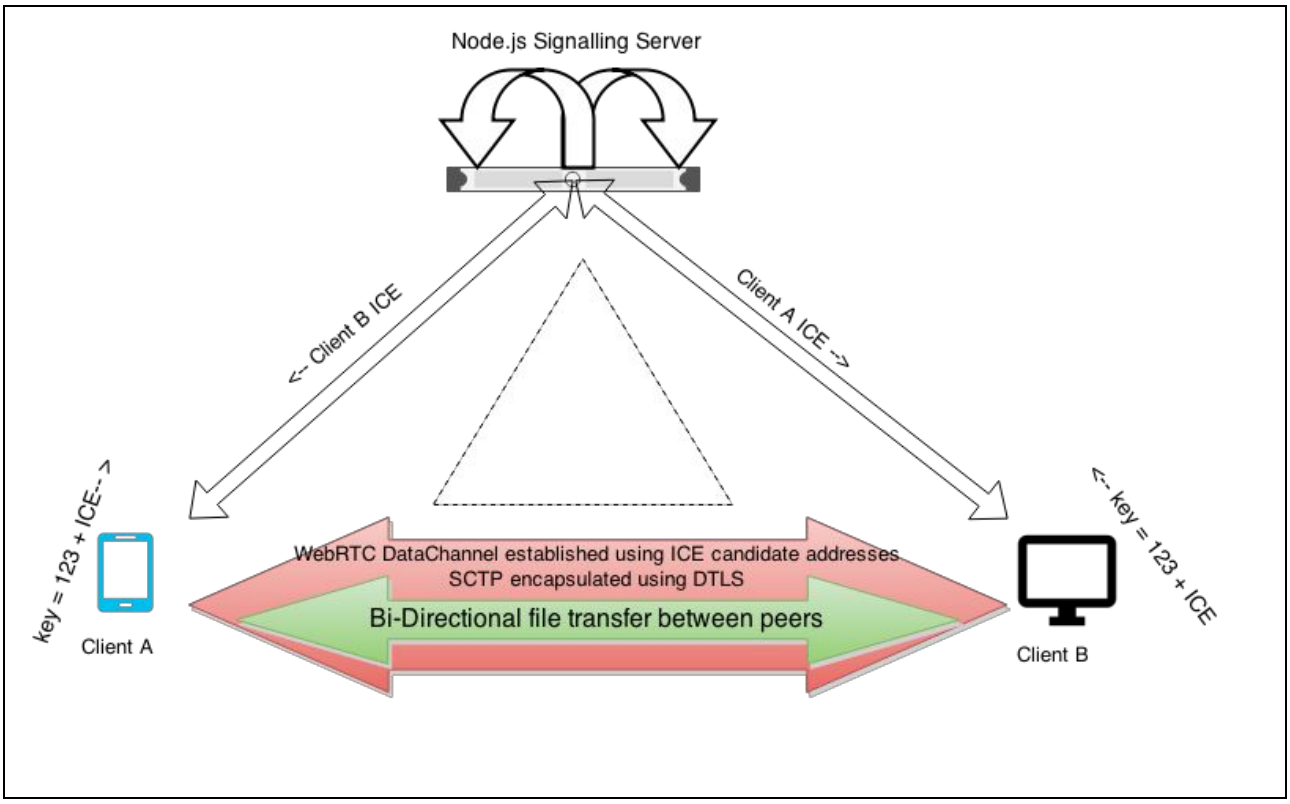
candidates are then sent to the NodeJs application server, to be eventually requested by a peer with a corresponding key.

A typical ICE candidate, which is sent to the nodeJS signalling server along with a key to identify the connecting peers:


```
{
        "type": "candidate",
        "mlineindex": 0,
        "candidate": "candidate:2070390239 1 udp 2122260223 192.168.1.134 43351 typ
host generation 0"
}, {
        "type": "candidate",
        "mlineindex": 0,
        "candidate": "candidate:4197426027 1 udp 1686052607 78.105.4.208 51114 typ
srflx raddr 192.168.1.134 rport 43351 generation 0"
}, {
        "type": "candidate",
        "mlineindex": 0,
        "candidate": "candidate:904157487 1 tcp 1518280447 192.168.1.134 0 typ host
tcptype active generation 0"
}
```


Signaling step-by-step walkthrough:

1. Both peers connect to signalling server: ***connect()***
2. Both Peers send their key + ICE information to the signalling server: ***sendMessage(info)***
3. Clients poll signalling server for ICE information using agreed key: ***poll(key)***
4. Signalling server sends ICE information to each corresponding peer: ***getMessages(info)***
5. NAT traversal 'hole punching' begins between peers (STUN / TURN) upon receipt of ICE candidate information
6. Peer connection established, providing NAT traversal and other obstacles are avoided.



The signaling server simply provides a message passing service for keys and temporarily holds ICE candidate addresses sent by two connecting peers which enabled them to fetch their respective ICE candidates. From this point on, the peers have enough information from the ICE candidate addresses to attempt to use these to establish a peer connection. This process of a peer to peer connection between them is purely a peer to peer affair defined and standardised by WebRTC in the ICE and STUN suit of protocols. From this point on, connection establishment is purely a peer to peer affair and does *not* involve a third party (including the signaling server). However, should the ICE candidate addresses totally fail, Traversal Using Relays (TURN) is required.

The TURN protocol provides a fallback in which a TURN server acts as a dumb relay between two peers. RFC5245 discusses the ICE protocol, which codifies the STUN and TURN protocols explaining their roles which collectively confront the challenge of connecting two hosts peer to peer when residing behind a NAT or similar obstructions.

The following code extract is an abbreviated form of the *connect()* function which peers call using the XMLHttpRequest object (ajax) as part of the signaling process. The framework for the function is taken directly from the Nodejs signaling server application which was adapted for this project, the basic framework of which was taken from the indispensable book “WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web” by Dr. Alan B. Johnston and Dr. Daniel C. Burnett.

The book explains many of the technologies applied in this application. Dr. Alan B. Johnston is an active participant in the IETF RTCWEB working group and an editor of the PeerConnection and getUserMedia W3C WEBRTC specifications which without their existence would not make this type of application possible within a web browser. The abbreviated connect function, is below:

```
// Copyright 2013-2014 Digital Codex LLC
// You may use this code for your own education.  If you use it
// largely intact, or develop something from it, don't claim
// that your code came first.  You are using this code completely
// at your own risk.  If you rely on it to work in any particular
// way, you're an idiot and we won't be held responsible.
// handle XML HTTP Request to connect using a given key
function connect(info) {
  var res = info.res,
          query = info.query,
          thisconnection,

          connectFirstParty = function() {
          if (thisconnection.status == "connected") {
          // delete pairing and any stored messages
          delete partner[thisconnection.ids[0]];
          delete partner[thisconnection.ids[1]];
          delete messagesFor[thisconnection.ids[0]];
          delete messagesFor[thisconnection.ids[1]];
          }
          connections[query.key] = {};
          thisconnection = connections[query.key];
          thisconnection.status = "waiting";
          thisconnection.ids = [newID()];
          webrtcResponse({"id":thisconnection.ids[0],
                  "status":thisconnection.status}, res);
          },
          connectSecondParty = function() {
          thisconnection.ids[1] = newID();
          partner[thisconnection.ids[0]] = thisconnection.ids[1];
          partner[thisconnection.ids[1]] = thisconnection.ids[0];
          messagesFor[thisconnection.ids[0]] = [];
          messagesFor[thisconnection.ids[1]] = [];
          thisconnection.status = "connected";
          webrtcResponse({"id":thisconnection.ids[1],
                  "status":thisconnection.status}, res);
          };
```

The *connect()* function simply takes each connecting peers' submitted key, putting these into an array of peers which wish to form a peer to peer connection with each other. A peer signaling channel is not established between the two pers until both have exchanged ICE candidates. When each peer has contacted the server with the same key the two peers are considered 'connected' and each peer can send their corresponding ICE candidates to the signaling channel. Fetching of the ICE candidates is achieved by each client polling the signaling server for ICE candidate messages from its future peer using the XMLHttpRequest object. Upon receiving the ICE candidates, peers being the STUN process of attempting to establish a true peer to peer connection.

Polling

Polling is the approach used for peers to request ICE candidate information from the signalling server. Polling commences after successfully calling the *connect()* function. The XMLHttpRequest object is used to call the Nodejs's sendMessage() function which is the mechanism by which each peer sends their ICE candidate information to the signalingServer. The corresponding 'getMessages()' function is then called, also via an XMLHttpRequest object to query the signaling server for ICE candidate information for the corresponding peer identified by their key.

sendMessage(info) function handler extract which receives ICE candidate information from peers, and stores these transport addresses according to each peers connection key.

When the signalling server is polled (via getMessages()) by a peer with a valid key, if the signalling server has ICE already received ICE candidates pertaining to that key, the ICE candidates pertaining to that key are returned to the connection client.

```
function sendMessage(info) {
 var postData = JSON.parse(info.postData),
          res = info.res;

  messagesFor[partner[postData.id]].push(postData.message);
  log("Saving message ***" + postData.message +
          "*** for delivery to id " + partner[postData.id]);
  webrtcResponse("Saving message ***" + postData.message +
                    "*** for delivery to id " +
                    partner[postData.id], res);
}
exports.send = sendMessage;
```

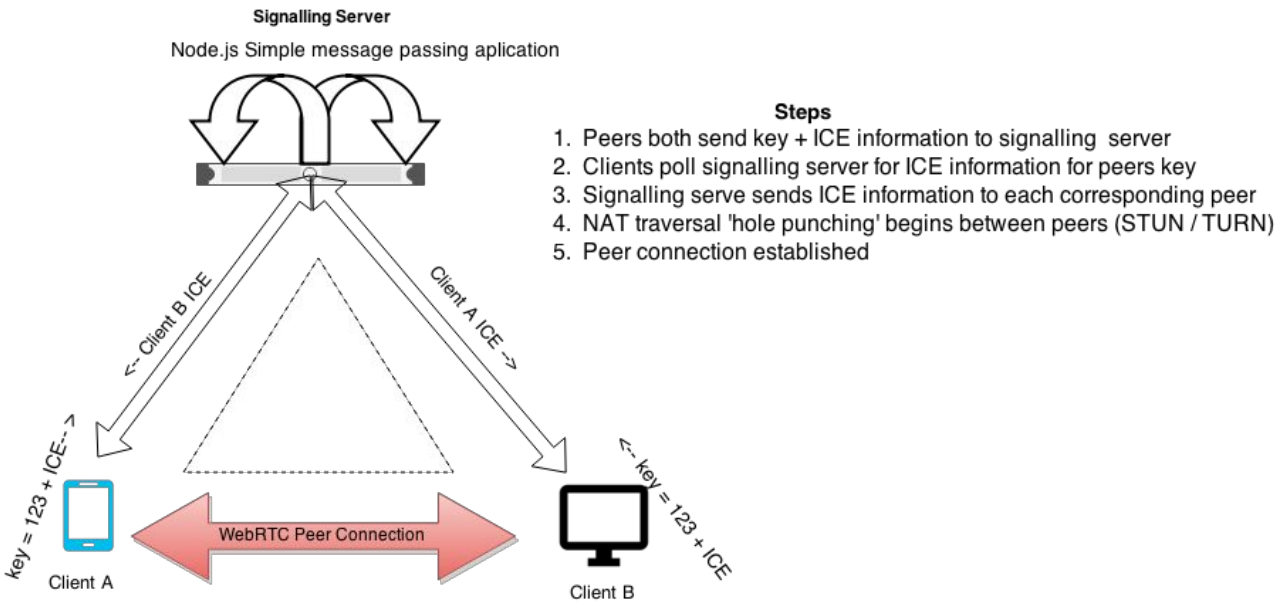
getMessages() Nodejs function extract which is polled by the client side poll(); function using Ajax to request ICE candidate information corresponding to the key which is sent at each poll() call.

```
function getMessages(info) {
  var postData = JSON.parse(info.postData),
          res = info.res;

  if (typeof postData === "undefined") {
          webrtcError("No posted data in JSON format!", res);
          return;
  }
  if (typeof (postData.id) === "undefined") {
          webrtcError("No id received on get", res);
          return;
  }
  if (typeof (messagesFor[postData.id]) === "undefined") {
          webrtcError("Invalid id " + postData.id, res);
          return;
  }

  /* log("Sending messages ***" +
          JSON.stringify(messagesFor[postData.id]) + "*** to id " +
          postData.id); */
  webrtcResponse({'msgs':messagesFor[postData.id]}, res);
  messagesFor[postData.id] = [];
}
exports.get = getMessages;
```

For legibility, the signalling process diagram is repeated below.



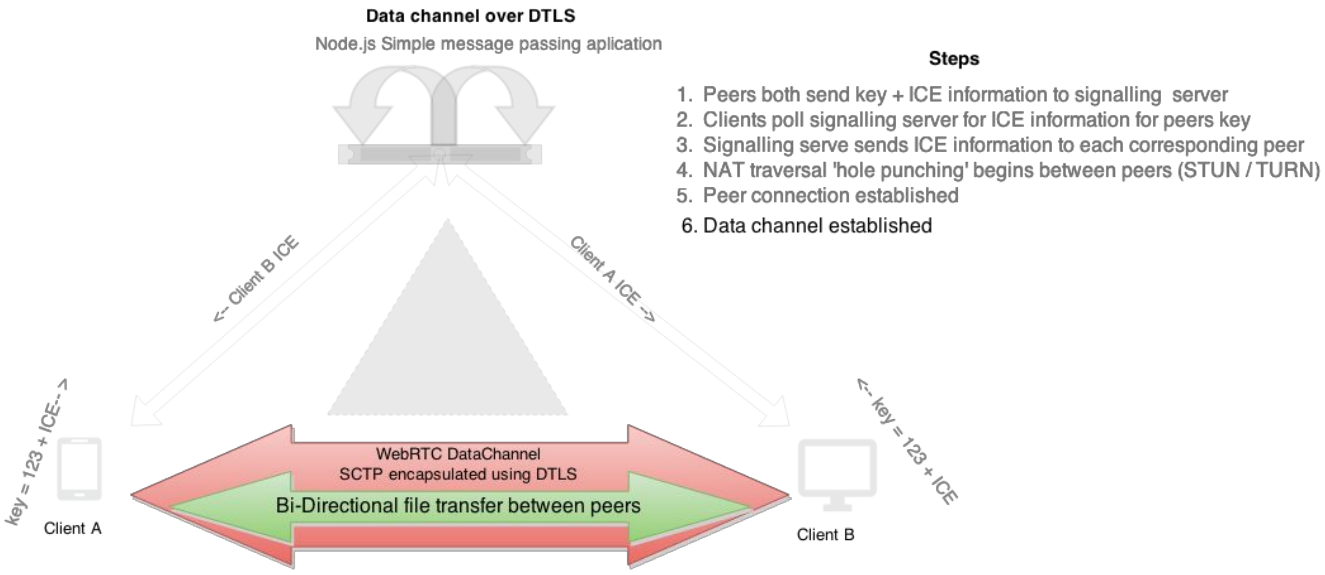
During development of the signaling server framework, its copyright disclaimer did not fail to live up to its brashness. The signaling server was first simplified for testing polling between just two peers rather than many as a proof of concept, following a more fundamental issue relating to the frameworks web server implementation causing it to consistently crash once the application grew above a certain size. This was addressed by investigating Nodejs's web server implementation from the sample framework.

Nodejs server also acts as the application server. This role is simply to fulfill requests for downloading of the web application, which may also be ran locally from a USB stick once downloaded. However, buffer sizes allocated to the web server in the framework proved too small which, although a minor change, considerable time was taken to identify this flaw. Though perhaps out-of-scope, this insight into web server development and memory management proved a useful reminder when addressing the applications constraints when interfacing with SCTP, discussed in the 'Datachannel' section of this paper. SCTP is the transport protocol used when peers transfer data.

Datachannel

Exchanging generic data between peers

Once signalling is complete, and the ICE process is complete, a direct peer to peer connection exists. The diagram below highlights the steps that are taken in order to successfully establish the peer to peer connection.



“The Stream Control Transmission Protocol (SCTP) is used in the WebRTC context as a generic transport service allowing WEB-browsers to exchange generic data from peer to peer. WebRTC uses the SCTP protocol for transport which is “encapsulated over DTLS” (Tuexen et al., 2015).

These peer connections are collectively known as “Data Channels” (Jesup, Loreto and Tuexen, 2015) which is currently an IETF Internet-Draft on the Standards Track.

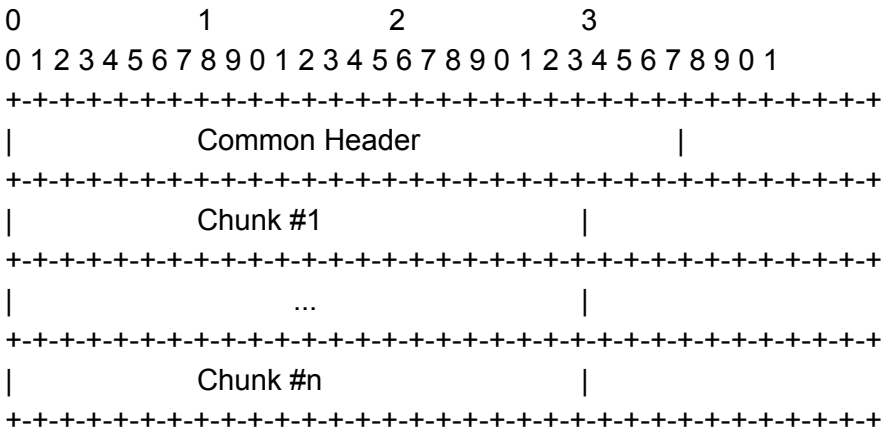
Following a successful signalling processes, the developed application further uses the WebRTC web APIs to extends its usefulness by using its peer connection functionality to allow two connected peers to send arbitrary data between each other directly from their own object stores. That is, the datachannel is combined with the applications’ IndexedDB object store to provide the mechanism for users to transfer their locally backed-up files between chosen peers.

As stated, Peer to Peer data transfer between peers is only achieved after the signalling process is successful, after which, data may be transferred between peers over the transport layer connection which uses the Stream Control Transmission Protocol (SCTP), which is encapsulated in DTLS. Once a file transfer is requested between peers, one SCTP packets

contains at most one chunk of the files locally stored in the applications IndexedDB store, which is then encapsulated in DTLS for private transportation over the link toward the receiving peer.

Composition of an SCTP packet

Understanding the composition of an SCTP packet matters because it influenced, and put constraints on the development of this application including throttling, and packet size limitations.
An SCTP packet, shown below, is comprised of the following headers and chunks: its 'Common Header', is much like its cousin TCP containing a peers' Source and Destination port number, a checksum, as well as a 'Verification Tag', the purpose of which is to "validate the sender of this SCTP packet" (RFC 2960):



Although "Multiple chunks can be bundled into one SCTP packet" (RFC 2960), this is limited to the MTU of the link, and caused complications in the development of the application.

- Chuck sizes had to be kept small (below 65K)
- Additionally, the throughput of of chunks sent to a peer had to be be limited to prevent browsers from terminating peer connections due to exceeding both Datachannel throughput restrictions implemented in User Agents implementations of the WebRTC datachannel, as well as filling the SCTP buffer too quickly which would cause the operating system to terminate these connections.

Troubleshooting and testing

Identifying the maximum SCTP Chunk size & throughput restrictions

Using file chunks to address SCTP message size limit of 65k

During the early development of the data transfer between peers feature, the development faced a barrier when sending large files peer to peer using the WebRTC client libraries. It was later identified this was due to the message size limit imposed by SCTP, as well as WebRTC not handling congestion events caused by sending data too quickly, and filling SCTP buffers between peers over the peer connection.

Google developer la...@google.com for the Peer Connection APIs in Chrome browser implementation explained this limitation as follows:
“When you go over 64k, you hit the SCTP protocol message-size limit. Specifically, there are only 16 bits for a given 'chunk'. We're looking into whether to document this as a limitation for a single send(), or put together some sort of fragmentation protocol.” (Issue 2270 Code.google.com, 2013). Note that a ‘chunk’ being referred to here is in relation to the maximum number of chunks an individual SCTP packet can trail following its Common Header. This is not in relation to the chunks of user files stored in the IndexedDB.

Addressing user agent imposed Datachannel throughput restrictions

Despite the SCTP protocols’ support for congestion management via control packets (RFC4960) the WebRTC libraries do not provide an interface to the developer when dealing with congestion notifications. Thus, the event of filling a peers’ SCTP buffer is not handled by the WebRTC APIs and the transport connection is simply terminated by the underlying operating systems’ networking stack.

This leaves the developer somewhat left in the dark initially when this occurs, given the lack of error handling in the APIs and documentation on this issue: A lot of time was wasted assessing and testing to identify the possible causes to SCTP connection terminations. Multiple web browsers were used, operating systems in effort to identify the cause.

A crude solution to avoid overfilling the SCTP buffers was implemented to address the flow control issue: A restriction on the size of data being sent, as well as an arbitrary delay between each chunk being sent was imposed. During research into the cause of this issue, throughput control and chunk-by-chunk sending proved to be the most reliable way to reduce the risk of sending too much data too quickly over the datachannel.

Code snippets which tackled this challenge directly are now presented and help explain the approach taken in this application to mitigate the SCTP flow control and packet size restrictions during upon research and recommendations taken from implementers of these protocols.

How the application addressed reliably sending files between peers

Mitigating SCTP limitations

Signposts from developers on the WebRTC signalling implementation for Google Chrome gave useful timesaving advice, which, can be attested to when the specifications for SCTP are reviewed which specify SCTP’s message size limits:

“orhiltch: Messages longer than a few (*cough* 16 *cough*) kb are unlikely to work well. SCTP doesn't have large buffers internally for storing these while

transmitting/fragmenting/defragmenting/etc. Please break them up into chunks when sending, and de-chunkify on the other side. It'll be much nicer to the memory pressure in chrome tab, and the system as a whole. It'll also scale quite well to much larger sizes." (Code.google.com, 2013)

Designing the application's Data Store (implemented using IndexedDB) around the restrictions imposed by SCTP made the transferring of files between peer connections much simpler as all files are first split into small chunks, and then stored into IndexedDB. The size of the chucks is configured to be well below the maximum permitted for SCTP.

Because of this, sending a file to another peer is made simpler given there is a guarantee that every chuck in the database is already small enough to be sent over an SCTP data channel connection and is unlikely to exceed throughput limitations: a crude *sleep()* is placed between each sending of a chunk.

What's more, the entirety of a file need not be stored nor by the same peer due to the enforced chunked nature on all objects stored in the IndexedDB database. One peer many only have the capacity to store part of a file, but can still contribute to the reforming of a file by sending relevant chunks to a peer which requests the file by referencing its fileId.

The code snippet below demonstrates use of the FileAPI (Tools.ietf.org, 2012) which is used to chunk all files on the fly as they are inserted into the offline IndexedDB storage thus limiting the size of chunks of all files stored within the IndexedDB object store before the need to transmit over SCTP.

```
var reader = new FileReader();
        reader.onload = function(file) {
                if ( reader.readyState == FileReader.DONE) {
                result = file.target.result;
                //Generate uuid for file
                var fileId = window.uuid();
                //Get file size
                var fileSize = result.byteLength;
                //Begin chunking...
                //Work out number of chunks needed
                var maxChunkSize = 64512; //bytes
                var maxChunkSize = 24512; //bytes
                var numChunksNeeded = Math.ceil(fileSize/maxChunkSize);

                var chunks = [];
                //Create blob from ArrayBuffer
                var blob = new Blob([reader.result], {type: window.fileType});
                reader.address = window.URL.createObjectURL(blob);

                 //Add chunks to indexedDB
                var start = 0;
```

```
var end = maxChunkSize;
db.transaction("rw", db.chunks, function() {
//Begin chunking
for(var i=0; i<= numChunksNeeded; i++)
{
var chunk = blob.slice(start,end);

db.chunks
        .add({
        fileId: fileId,
        boxId: window.boxId,
        fileName: window.fileName,
        fileType: window.fileType,
        chunkNumber: i,
        numberOfChunks: numChunksNeeded,
        chunkSize: window.fileSize,
        chunk: chunk
        });
console.log("uuid: " + fileId);
//Increment start by maxChunkSize
start = start + maxChunkSize;
end = end + maxChunkSize;
}//End create chunks
```

FileReader() “provides methods to read a File or a Blob, and an event model to obtain the results of these reads.” (Rescorla and Modadugu, 2015). This FileReader interface is used for the loading of user files from their personal filesystem into the IndexedDB object store.

Chunking and storing: Manipulating Blob objects

For clarification, the terms may need defining. The FileAPI defines a ‘Blob’ as the following: “A Blob object refers to a byte sequence, and has a size attribute which is the total number of bytes in the byte sequence, and a type attribute, which is an ASCII-encoded string in lower case representing the media type of the byte sequence.” (Rescorla and Modadugu, 2015)

An ‘ArrayBuffer’ object is defined: “to represent a generic, fixed-length raw binary data buffer.” (Mozilla Developer Network, 2014).

Upon loading a file into the application, the ‘onload’ event is fired by the FileReader interface. This is caught, and then the process of chunking the file, and storing each chunk one by one into the object store is performed. This is achieved by calculated the number of chunks needed based on the file size and maximum chunk size. (var numChunksNeeded = Math.ceil(fileSize/maxChunkSize);).

The application does not interface with the IndexedDB API directly, instead the ‘Dexie’ framework is used which is “A Minimalistic Wrapper for IndexedDB” (Dexie.org, 2015). This simplified development, such as being able to interface cleanly with IndexedDB transaction mechanisms for data integrity. That is, each chunk is only stored into the database in a transactional fashion and therefore should and errors occur, the insert transaction is automatically rolled back. This is done by means of a read/write lock on the IndexedDB database for each chunk insertion: db.transaction("rw", db.chunks, function() {...

Transferring ArrayBuffer chunks to peers over the Datachannel

When a user opts to send a file to their connected peer, in this current version of the application they must first click the Show my files button, then the ‘Send’ button next to the corresponding file they wish to send as follows:

The latter commences the following required steps in order to successfully send the file to a peer over the established datachanned:

- Retrieve each chunk pertaining to the users chosen file from the IndexedDB using its ‘fileId’
- Gather each chunks meta data.
- Communicate both the chunk payload, and its corresponding meta data to the peer in a reliable fashion

File meta data

Each chunk object stored within the storage layer of the application does not just store the raw chunk of all files submitted. It also contains, and is necessary to keep track of the details of these chunks. This is done by applying metadata describing each chunk when it is places into the database. When a chunk is stored into IndexedDB, an object is created created both containing the raw chunk itself, a well as other useful properties for identification:

```
chunk = {
        boxId: window.boxId,                    //A UUID text string to uniquely identify the ‘box’
        chunk: blob,                                    //Blob containing raw chunk of file
        chunkNumber: window.curChunk.chunkNumber,    //Integer
        fileId: window.curChunk.fileId,         // A UUID text string to uniquely identify the file
        fileName: window.curChunk.fileName,             //Text string
        fileType: window.curChunk.fileType, //Text strign of the Mime filetype
        numberOfChunks: window.curChunk.numberOfChunks // Integer
}
```

This meta data is useful when transferring when transferring files between peers, as it provides the possibility to know the total number of chunks ahead of receiving every chunk which may help estimate download time. Additionally, more fundamental benefits are provided by the chunk number which is used for file reassembly at the destination peer. Without this, is it would be extremely challenging to reconstruct a peers file once send to another peer.

Communicated Chunk Meta Data to the peer

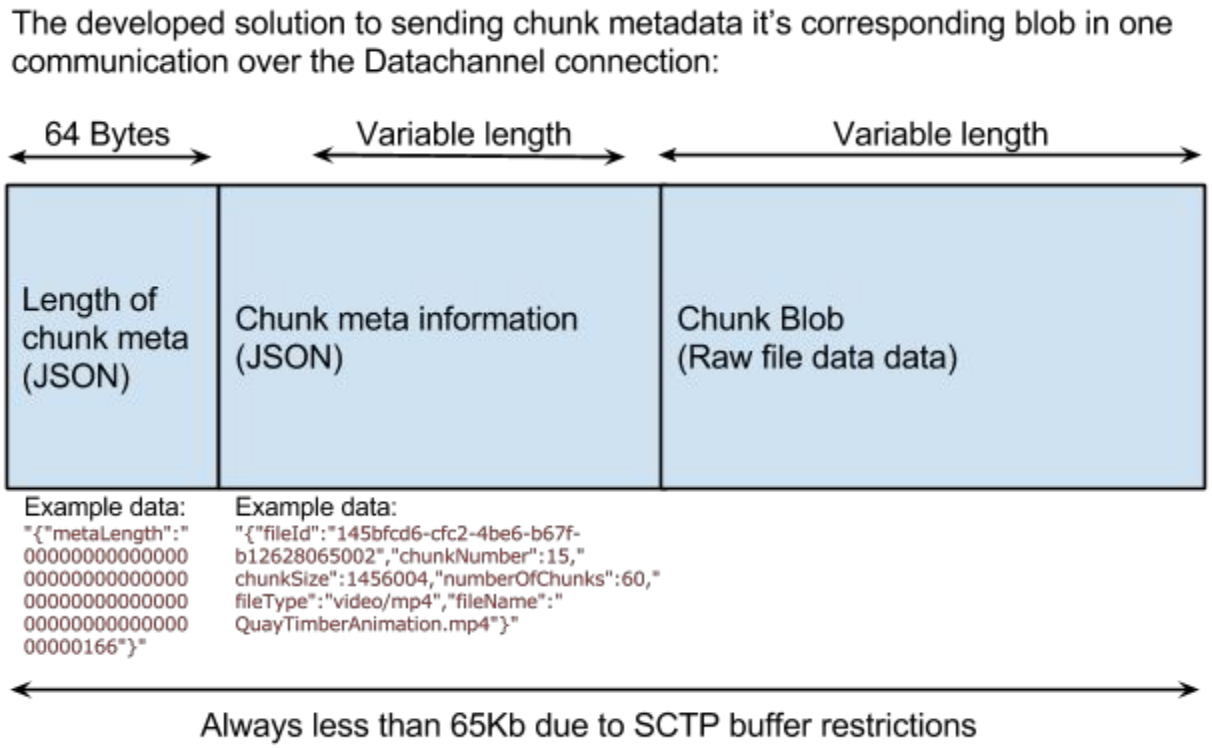
A mechanism was developed to communicate this chunk meta data to the peer. Two approaches were considered:

- A stateful system: Whereby the sending peer would coordinate with the receiving peer by sending it ‘control packets’ allowing the receiver to know what chunk meta to apply to the next received chunk ahead of time. Thereby separating the communication of chunk meta and chunk data.
- Combining the chunk meta data and chunk Blob into one communication by implementing a standard header containing this meta data to be prepended to every ArrayBuffer sent.

The second approach was adopted for the transference of files between peers because it was deemed to add unnecessary complication to introduce such tight coupling between sending and receiving peers with a state-like system. Additionally, future development of the application may see peers receiving data from more than one peer at the same time; this would make the first approach complex to manage due to the need to keep track of what meta data to apply to which incoming chunk from which peer.

Instead, the application is written to respond to a send file request by first extracting both chunk metadata and it’s corresponding Blob chunk and combines these two bits of data into a single ArrayBuffer which is then sent across the Datachannel in one communication.

The format of this communication can be visualised as a packet in the following way:



As the implemented solution infers, for both peers to be able to send and store file chunks meaningfully to each other, it is both the sending and receiving peers' responsibility to ensure this consistent method of representing chunks and their various metadata is adhered to.

By taking this approach, flexibility is introduced into the program: By placing the chunk Blob at the end of the payload, this allows the file chunks to be of arbitrary length, (so long as the sum of the entire payload remains below 65Kb as explained). Such design considerations may benefit the applications future adaptability to changes to the WebRTC specifications. For example, the 65Kb limit on message sized may be increased in the future: "We don't plan to support larger packets until the changes to do message splitting make it into the SCTP spec." (Code.google.com, 2013) thus, designing the Datachannel communication such as it is explained above allows for growth in the size of this packet.

This approach also allows for the fact that chunk metadata may change in size. For example: small file names will reduce the memory required for the chunk metadata information increasing efficiency when transporting many large numbers of chunks for large files. However, memory saved by this is arguably marginal.

The abbreviated following code extract shows the implementation of this message structure, first as leaving a peer, and then the stripping of the 'header' from the message structure on its receipt.

1. *sendChunksToPeer(e)* Catches the event of a user clicking the ‘Share’ button next to a file name.

2. The chunk is then queried from the database by its fileId.

3. In preparation for the message, the total chunk metadata is read directly from the chunks’ IndexedDB entry.
4. The length in bytes of this chunk metadata is calculated for the header of this communication
5. A new Blob is constructed, formed out of the meta data length calculation, the chunks’ metadata itself and the Blob file chunk.
6. The newly created Blob is read back into the FileReader interface, which causes the Blob to be of type ‘ArrayBuffer’. This is a necessary type conversion given Google Chrome does not yet support the transfer of Blob types over Datachannel connections (mmr, Code.google.com, 2014), though FireFox does
7. This new ArrayBuffer which is now headed with the needed metadata and ready to be sent over the datachannel connection.
8. dc.send(result = file.target.result); Sends the ArrayBuffer to the client containing all required data in the prescribed format.
9. A crude delay is used to throttle sending of chunks between each iteration to avoid filling the SCTP buffers and terminating the connection. A more elegant solution would use closures to more accurately throttle the connection.

```
function sendChunksToPeer(e) {
	//Get file id:
	var fileId = e.target.dataset.fileid;

	db.transaction('r', db.chunks, function() {

	db.chunks.where("fileId").equals(fileId).each(function(chunk) {
		//Transaction scope

		//Sending file meta...
		var meta = {"fileId":chunk.fileId, "chunkNumber":chunk.chunkNumber,
"chunkSize":chunk.chunkSize,
"numberOfChunks":chunk.numberOfChunks,"fileType":chunk.fileType,"fileName":chunk.fileNa
me};
		var lengthOfMeta = JSON.stringify(meta).length;
		lengthOfMeta = zeroFill(lengthOfMeta, 64);
		var metaLength = {"metaLength":lengthOfMeta}; //Always 81 characters when
stringified
		var header = JSON.stringify(metaLength) + JSON.stringify(meta);
		var sendChunk = new Blob([header, chunk.chunk]);
		url = window.URL.createObjectURL(sendChunk);
		//Needs to be sent as an arrayBuffer
		var reader = new FileReader();
			reader.onload = function(file) {
```

```
                    if( reader.readyState == FileReader.DONE ) {
                            for(var i=0;i<=99999999;i++) {}//Crude delay!
                            dc.send(result = file.target.result);
                    }//End FileReader.DONE
            }//End reader.onload
            reader.readAsArrayBuffer(sendChunk);
```

On receiving a message of type ArrayBuffer, the receiving peer processes all ArrayBuffers in the same way. First, by stipping off the metaLength field.

1. Next the chunk meta data can be extracted given the metaLength is given.
2. The metadata for the chunk is that parsed using JSON.parse(result);
    a. This is eventually used as the metadata for the chunk when it get stored into the object store.
3. The raw data of the file's chunk, of type Blob is extracted from the received ArrarBuffer by calculating the header offset, and performing the *slice()* method at that point: var *chunkBlob = blob.slice(headerOffset);* //Get chunk payload.
4. Finally the chunk itself is added to the IndeedDB object store by passing the chunk and metadata to addChunkToDb(chunkBlob). Note, the chunk metadata is accessed via the window.curChunk object which is globally accessible allowing for easy update of the screen to indicate progress to the user.

```
//Get length of file meta (size specified is always a zero filled 64byte long string at the beginning of the blob
            var metaLength = blob.slice(0,81);
            //Read metaLength to get size in bytes of chunk fileMeta
            var reader2 = new FileReader();
            reader2.onload = function(file2) {
            if ( reader2.readyState == FileReader.DONE ) {
                    result2 = file2.target.result;
                    var fileMetaLengthObj = JSON.parse(result2);
                    var fileMetaLength = parseInt(fileMetaLengthObj.metaLength);
                    window.fileMetaLength = fileMetaLength;
                    console.log("Meta length is:" + fileMetaLength);

                    //Split file meta from beginning of chunk
        var chunkFileMeta = blob.slice(81, window.fileMetaLength + 81); //First get file type, chunk number etc
                            var reader = new FileReader();
                            reader.onload = function(file) {
                            if ( reader.readyState == FileReader.DONE ) {
                                    result = file.target.result;
                                    if ( result.length > 0 ) {
                                            window.curChunk = JSON.parse(result);
                                            //End extract file meta from blob
```

```
                                    }//End check read data is > 0

                                    //Start send data payload
                                    var headerOffset = 81 + window.fileMetaLength;
                                    var chunkBlob = blob.slice(headerOffset); //Get chunk
payload
                                        //Store chunkBlob into IndexedDB
                                        addChunkToDb(chunkBlob);
                                }//End reader.readtState == DONE
                            }//End reader.onload
                            reader.readAsText(chunkFileMeta);
                            //End extract file meta from blob
                }//End IF reading byte length of file Meta
                }//End get bytelength of fileMeta

                reader2.readAsText(metaLength);
```

Privacy & transport reliability at the transport layer

Though this development does not aim to provide, nor promote this application as a secure file transfer and backup application, the technologies it uses do provide privacy protecting features. Especially at the transport layer during the entire duration of a peer to peer session: “SCTP can be used on top of the Datagram Transport Layer Security (DTLS) protocol” (Tuexen et al., 2015). This has the added benefit of providing a level of privacy to peers transferring files between each other given, “DTLS over SCTP provides communications privacy for applications that use SCTP as their transport protocol and allows client/server applications to communicate in a way that is designed to prevent eavesdropping and detect tampering or message forgery. (RFC 6083).

The sending of data between peers therefore has a level of privacy for the users due to STCPs’ encapsulation over Datagram Transport Layer Security (DTLS). “The DTLS protocol provides communications privacy for datagram protocols.” (RFC 6083). DTLS is used over TLS encryption because WebRTC also permits the establishment of unreliable transport data over UDP-like channels, which results in a more flexible and expandable application.

For example, to support video conferencing and other lossy media, a reliable transport is unnecessary as it may frustrating delays to real-time media: “A real-time communication session cannot pause for a second or two … while awaiting the retransmission of missing information” (Johnston et al, 2014 p.226).

DTLS, on the other hand, was designed to secure unreliable transports, given that “TLS must run over a reliable transport channel -- typically TCP [TCP]”(RFC 6083). Therefore because of WebRTC data channel adoption of DTLS to encapsulate its data channels, both reliable and unreliable peer connections can be established. WebRTC builds its datachannel using SCTP

protocol, largely because of this need to be flexible in supporting both reliable and unreliable transports.

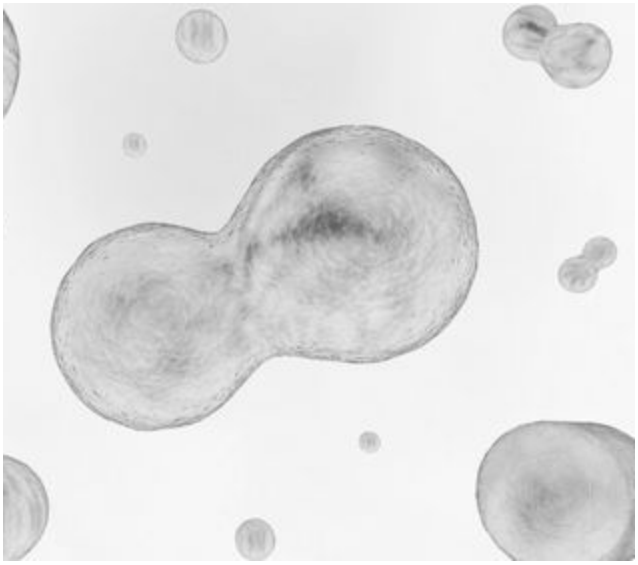
For the development of this application, naturally, the default reliable transport offered by SCTP was used for the transmission of files between peers to promote data integrity. This resulted in being able to support both the reliable transfer of files, and potentially expand the application further to support loss tolerant video streaming capabilities in the future. Johnston points out: “the reliable or semi-reliable delivery option is one that is extremely useful for web developers” (Johnston et al 2014, p.226).

Peer Exchange

Small world exchange

The program implemented does not aim, (nor is it thought necessary) to seek a utopian perfectly scaling, fully addressable, load balancing and free-rider exempt peer2peer network. It did however fail to complete the overlay network which would add another layer of autonomy to the application. This would allow, a program which focuses on allowing peers that wish to, establish their own swam which may or may not grow organically through the sharing of the swarms’ identifier though out of bound communications. This in itself is scalable, especially when motivations to share are considered: because members of a smaller group are vested in sharing data for which only they are interested in, and thus willing to commit resources, else why would a peer join the group?

The latter is comparable to a ‘network of trust’, though only to the extent that there is an element of control over the initial members of a group. We move away from this analogy abruptly in allowing these ‘small worlds’ to connect to each other and form a larger group. That is, via out of band communications, and subsequently successful signalling, a small world group *may* connect with another small world group forming a larger swarm organically. Visually this might be best represented by the term ‘cell division’ stolen from biology.



The image to the left is representative of two smaller groups having connected to form a larger group.

Before establishing a connection between the two groups, an out of band communication to the signaling server overlay network is needed to allow session establishment between all nodes in the two groups.

Evaluation and Conclusions

Product evaluation.

An evaluation of the product, focusing on its build quality and fitness for purpose. This should include, and may be structured around, an assessment of the extent to which the product meets its requirements and other criteria set out in the terms of reference. Results of any user or client evaluation should be taken into account.

Visually this product is not successful, however, the extent to which the application meets its the criteria it set out meet is now scrutinised.

The application aimed to facilitate:

1. A reliable, unlimited persistent storage mechanism which worked within a web browser.
2. Reliable session establishment between peers
3. Reliable transfer of data between those peers (in a peer to peer fashion)

Overall product fitness for purpose

A reliable, unlimited persistent storage mechanism has been realised over the development of this project. It can be used to store arbitrary data with in the web browser and, which can further be queried and additional items added. Crucially, this stored data can also be sent between peers in a peer to peer fashion. Critically, therefore this aim has been met.

However, the project did fall short, by not producing a fully CRUD capable system with a peer overlay network. Whilst it *is possible to* manually delete items from the IndexedDB store, there has been no user friendly mechanism programmed to reach this end.

Session establishment

Reliable session establishment, has been achieved by gaining an understanding of an array of various network and web technologies (WebRTC, ICE, TURN and STUN). This research made it possible to tackle and attempt to understand the larger issues surrounding peer to peer connectivity over the internet and implement standards based approaches to resolving these issues. The end result was a signaling server which supports the marrying of two peers allowing them to eventually form a peer connection- a prerequisite to being able to transfer data between peers

Reliable transfer of data between peers

Implementing reliable data transfer between peers was the most significant development task after signalling due to SCTP restrictions and troubleshoot attempting to resolve these challenges. Many failed prototypes and misguided attempts at ‘coding around the issue’ took much development time, however the end result was a more resilient product: For example, the new understanding that ArrayBuffer transfers over Datachannel connections are more commonly supported that raw Blog transfers made the product cross-compatible with more web browsers. Additionally, the development for correct handling of file chunk metadata lead to the development of a more resilient method of transferring data between peers which was throttled, and did not exceed operating system imposed buffer size limits by limiting the size of chunks send over peer connections.

Process evaluation & Conclusions and recommendations

It is felt that a significant amount of self reflection on performance, project management and ability took place throughout the duration of the project. This led to an even appreciation of tools, such as GIT for version control management. Though more importantly the value in sharing of ideas which became clear through researching the history of WebRTC and the stakeholders.

Management of the project was sporadic yet activity on it remained largely consistent. This is likely reflected in the GIT version control logs.
A significant amount of learning took place during this project, and appreciation for technology as a whole which solidified the idea that it’s never obvious what tools, experiences or ‘knowledge’ may be of use on a project.

Finally, it became clear a significant lack of good understanding over programming practises hindered the quality of the development. A deeper appreciation of protocol design, in hindsight may have aided understanding over many of the topics brought forward by this project’s application. This is arguably due to the strong history of real time communication has in telephony networks, as much of the literature alludes to this era. As such, telecommunications is discussed in good detail in Johnstons’ WebRTC APIs and RTCWEB Protocols Work (Johnston, 2014) which explains signaling in the context of public switched telephone networks (PSTN), which gives perspective over the historic development of the technologies.

The aims and objectives of the project remained at the forefront of the development despite often the words to describe what was being created becoming, at times, unclear though despite this the learning process continued. For example, a key milestone was the consent of out-of-band communication and the role it plays is Signalling. Prior to the project such vocabulary simply didn’t exist, though as a deeper understanding of the problem space increased so did the ability to understand and then articulate to others the task and challenged at hand.

This disparity between the aims and objectives of the project and the end result can be seen clearly in the handbook: The definition and aims of the project are perhaps naively simple and unconsidered. It is this naivety which conceivably allowed the project to loose sight of its original objectives in creating what is traditionally perceived to be a peer to peer network. For example, no peer overlay network was developed for this application. Despite this, what emerged from it was an alternative viewpoint from which to observe what *peer to peer* network could mean, and weather more loosely couples peer to peer networks have a place alongside utopian self scaling, addressable peer to peer networks.

Conclusions and Recommendations.

The web has become a difficult thing to define as it has evolved to mean many things to many people. It is debatable whether this application has done nothing but to do a complete full circle from what the first website set out to achieve: an “information retrieval initiative aiming to give universal access to a large universe of documents.” (Berners-lee, n.d.). The DRY principle is another reason for this project not to exist: FTP still serves many well as a means to transfer files. Is it then just a matter of interface and availability of it? This is perhaps the strongest argument for technologies such as WebRTC which tie themselves so closely to the web browser. The web browser is a common interface, which isn’t intimidating to a lot of people. This makes the barrier to entry low. Many other technologies can of course support a richer array of media; music, video, conferencing. Yet, often they take the common interface of a web browser causing disconnection and incompatibility run rife. At the risk of concluding on a duff note perhaps what has changed more significantly over the web is its willingness to be interconnected. There are efforts to do this, such as the Resource Description Framework (RDF) “a framework for representing information in the Web” (Cyganiak and Wood, 2014). What this project hopes to have demonstrated, however, is the potential that multi stakeholder developed technologies bring such as WebRTC, have to make possible interesting software experiments which can be enjoyed and push development forward to test richer and more interesting applications.

Next work

The next aim for this project is the completion of the signaling overlay network as the next stage for development of this application. Aswellas a more agreeable user interface.

References

Goodson, S. (2012). If You're Not Paying For It, You Become The Product. [online] Forbes. Available at: http://www.forbes.com/sites/marketshare/2012/03/05/if-youre-not-paying-for-it-you-become-the-product/ [Accessed 20 Apr. 2015].

Homan, R. (2001). The principle of assumed consent: the ethics of gatekeeping. *Journal of Philosophy of Education*, *35*(3), 329-343.

Gibbs, S. (2014). Facebook apologises for psychological experiments on users. [online] the Guardian. Available at: http://www.theguardian.com/technology/2014/jul/02/facebook-apologises-psychological-experiments-on-users [Accessed 20 Apr. 2015].

Hickson, I. (2010). Web SQL Database. [online] W3.org. Available at: http://www.w3.org/TR/webdatabase/ [Accessed 20 Apr. 2015].

Hickson, I. (2013). Web Storage. [online] W3.org. Available at: http://www.w3.org/TR/webstorage/ [Accessed 20 Apr. 2015].

Yasuda, K. (2015). Quota Management API. [online] W3c.github.io. Available at: https://w3c.github.io/quota-api/ [Accessed 20 Apr. 2015].

Mozilla Developer Network, (n.d.). IndexedDB. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API [Accessed 20 Apr. 2015].

Mehta, N., Sicking, J., Graff, E., Popescu, A., Orlow, J. and Bell, J. (2015). Indexed Database API. [online] W3.org. Available at: http://www.w3.org/TR/IndexedDB/#introduction [Accessed 20 Apr. 2015].

Ranganathan, A. and Sicking, J. (2015). File API. [online] W3c.github.io. Available at: https://w3c.github.io/FileAPI/#slice-method-algo [Accessed 20 Apr. 2015].

Code.google.com, (2013). Issue 2270 - webrtc - Sending 1169 byte message fails using Chrome with SCTP data channels enabled - Web-based real-time communication - Google Project Hosting. [online] Available at: https://code.google.com/p/webrtc/issues/detail?id=2270 [Accessed 20 Apr. 2015].

Johnston, A.B., Burnett, D.C., 2014. WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web, Third Edition, 3 edition. ed. Digital Codex LLC, St. Louis, MO.

Bergkvist, A., C. Burnett, D., Jennings, C. and Narayanan, A. (2015). http://www.w3.org/TR/webrtc/#h-peer-to-peer-data-example. [online] W3.org. Available at: http://www.w3.org/TR/webrtc/#widl-RTCPeerConnection-onicecandidate [Accessed 21 Apr. 2015].

Business.twitter.com, (2015). Twitter Tweet Engagement | Twitter for Business. [online] Available at: https://business.twitter.com/solutions/tweet-engagements [Accessed 22 Apr. 2015].

Jesup, R., Loreto, S. and Tuexen, M. (2015). draft-ietf-rtcweb-data-channel-13 - WebRTC Data Channels. [online] Tools.ietf.org. Available at: https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-13 [Accessed 22 Apr. 2015].

Tuexen, M., Stewart, R., Jesup, R. and Loreto, S. (2015). draft-ietf-tsvwg-sctp-dtls-encaps-07 - DTLS Encapsulation of SCTP Packets. [online] Tools.ietf.org. Available at: https://tools.ietf.org/html/draft-ietf-tsvwg-sctp-dtls-encaps-07 [Accessed 22 Apr. 2015].

Tools.ietf.org, (2015). RFC 6083 - Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP). [online] Available at: http://tools.ietf.org/html/rfc6083 [Accessed 22 Apr. 2015].

Tools.ietf.org, (2000). RFC 2960 - Stream Control Transmission Protocol. [online] Available at: https://tools.ietf.org/html/rfc2960 [Accessed 22 Apr. 2015].

Code.google.com, (2013). Issue 2270 - webrtc - Sending 1169 byte message fails using Chrome with SCTP data channels enabled - Web-based real-time communication - Google Project Hosting. [online] Available at: https://code.google.com/p/webrtc/issues/detail?id=2270 [Accessed 22 Apr. 2015].

Tools.ietf.org, (2007). RFC 4960 - Stream Control Transmission Protocol. [online] Available at: https://tools.ietf.org/html/rfc4960 [Accessed 22 Apr. 2015].

Rescorla, E. and Modadugu, N. (2015). File API. [online] W3c.github.io. Available at: https://w3c.github.io/FileAPI/ [Accessed 23 Apr. 2015].

Dexie.org, (2015). Dexie.js - Minimalistic IndexedDB Wrapper. [online] Available at: http://www.dexie.org/ [Accessed 23 Apr. 2015].

Mozilla Developer Network, (2014). ArrayBuffer. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer [Accessed 23 Apr. 2015].

Mmr, Code.google.com, (2014). Issue 3097 - webrtc - Sending of Blob fails in Chrome 33 - Web-based real-time communication - Google Project Hosting. [online] Available at: https://code.google.com/p/webrtc/issues/detail?id=3097 [Accessed 23 Apr. 2015].

Berners-lee, T. (n.d.). The World Wide Web project. [online] Info.cern.ch. Available at: http://info.cern.ch/hypertext/WWW/TheProject.html [Accessed 23 Apr. 2015].

Cyganiak, R. and Wood, D. (2014). RDF 1.1 Concepts and Abstract Syntax. [online] W3.org. Available at: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ [Accessed 23 Apr. 2015].