Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2016

Pedro Miguel Leite Ferreira Margarido

Distribuição de conteúdos multimédia na Web/P2P - SeedSeer

Distribution of multimedia content through the Web/P2P - SeedSeer

Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2016

Pedro Miguel Leite Ferreira Margarido

Distribuição de conteúdos multimédia na Web/P2P - SeedSeer

Distribution of multimedia content through the Web/P2P - SeedSeer

"Rather a mind opened by wonder, than one closed by belief."

— Gerry Spence

Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2016

Pedro Miguel Leite Ferreira Margarido

Distribuição de conteúdos multimédia na Web/P2P - SeedSeer

Distribution of multimedia content through the Web/P2P - SeedSeer

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Diogo Nuno Pereira Gomes, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor João Paulo Silva Barraca, Professor Auxiliar Convidado do Departamento de Eletrónica Telecomunicações e Informática da Universidade de Aveiro.

agradecimentos / **acknowledgements**

There are a number of people without whom this thesis might have never seen the light of day and to whom I am greatly indebted. I would like to thank my advisors, my family and my friends for all the patience while this thesis was still being developed and written. I cannot stress enough how their efforts to keep me focused to finish this thesis were hard on their part and how discussions with them led this work further day by day. Dealing with my own expectations of what SeedSeer should have been was certainly what took me so long to finish this thesis, I have to thank so much to everyone that pulled me out of this deadlock several times. Lastly, I want to thank my "online friends"that in this past year have been with me everyday and whose support kept me going. There’s nothing I value more than privacy, while I avoided being specific, I believe that everyone who was important to me while this thesis was being developed knows exactly how thankful I am for having them.

Palavras Chave HTML, WebRTC, P2P, WebSockets, BitTorrent.

Resumo

Desde a criação da Internet que existem inumeras formas de partilhar ficheiros, mas até ao dia de hoje é discutível se alguma possa ser considerada a melhor. A apetência do público em geral para conteúdo multimedia levou ao aparecimento de novas plataformas de distribuição de conteúdo como o Google Play, Netflix, Apple Store, entre outros. Estes conteúdos são distribuídos de forma centralizada e levam a grandes custos de infra-estrutura para essas entidades. Por outro lado, as redes P2P permitem a distribuição de conteúdos de forma descentralizada e com baixos custos, estes contudo, exigem aplicações específicas e conhecimentos técnicos, o que se torna uma barreira entre o consumidor e os conteúdos que estão disponíveis nestas plataformas. Nesta tese é desenvolvido um protótipo de uma nova solução, usando novos standards HTML5 como WebSockets e WebRTC para introduzir uma nova perspectiva de como os utilizadores podem partilhar e consumir conteúdo. Em termos simples, a abordagem desta tese procura trazer a rede BitTorrent para os Browsers usando apenas javascript, tirando partido da sua facilidade de utilização por não exigir qualquer tipo de instalação necessária. Usando WebRTC esta tese foca-se em como fazer crescer a rede dos Browsers de forma descentralizada, incentivando o consumo de conteúdo em comunidades de utilizadores num esforço para aumentar a privacidade e resistência à censura, assim como mitigar limitações de escala da solução. Os resultados deste trabalho demonstram que alguns conceitos utilizados nesta tese têm vantagens únicas que são relevantes para o público em geral, no entanto, estas vêm com o custo de algumas limitações que são inerentes e devem ser mitigadas.

Keywords HTML, WebRTC, P2P, WebSockets, BitTorrent.

Abstract

Since the inception of the Internet there are a lot of ways to share files, but still to this day it is arguable if there's a best one. The palatability of the general public for multimedia content created the need for new platforms of content distribution like Google Play, Netflix, Apple Store and some others. Contents that are distributed in a centralized way and that lead to great infrastructure costs to these entities. On the other hand, P2P networks allow the distribution of content in a decentralized way with low costs, these however require specific applications and technical knowledge, which is a barrier between the consumer and the contents that are available in these platforms. In this thesis a prototype of a new solution is developed, using upcoming HTML5 standards like WebSockets and WebRTC to introduce a new perspective to how users can share and consume content. In simple terms, the approach of this thesis is to bring the BitTorrent network into the browsers using only javascript, taking advantage of its ease of use by not requiring any kind installation. Using WebRTC this thesis focused in how to grow the browser's network while being decentralized, encouraging content consumption in communities of users in an effort to increase privacy and resilience to censorship as well as mitigate scaling limitations of the solution. Results of this research demonstrate that some concepts used in this thesis have unique advantages that are relevant to the general public, however they come at the cost of some inherent limitations that should be mitigated.

Contents

List of Figures

List of Tables

Abbreviations

API	Application Program Interface
BBS	Bulletin Board System
BLOB	Binary Large Object
BT	BitTorrent
CDN	Content Delivery Network
DHT	Distributed Hash Table
DMCA	Digital Millennium Copyright Act
DNS	Domain Name System
FAQ	Frequently Asked Questions
HTTP	Hypertext Transfer Protocol
ICE	Interactive Connectivity Establishment
IETF	Internet Enginerrring Task Force
IM	Instant Messaging
IP	Internet Protocol
IRC	Internet Relay Chat
JSON	JavaScript Object Notation
NAT	Network Address Translation
P2P	Peer-to-peer
SDP	Session Description Protocol
SIP	Session Initiation Protocol
STUN	Session Traversal Utilities for NAT

TCP	Transmission Control Protocol
TLS	Transport Layer Security
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
WebRTC or WRTC	Web Real-Time Communications
WS	WebSockets
XLM	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Chapter 1

Introduction

In the last couple of years the Internet has changed the way we communicate, from the way we do business to the way we socialize. It's growth has been huge to the point that, in 2015, the Internet has 3,2 billion of estimated users worldwide [4, 2], where the Internet penetration in developed countries is estimated to be as high as 82,2%. [2] In recent years the Internet became more mobile, as the range of devices that connect to it continue to grow everyday, including smartphones, tablets and Internet-of-Things devices. It has also become more social, with the growth of Social Network Services (SNS) changing the way most people communicate with each other and the way content is created and shared online [36, 17].

Even before the Internet was born back in 1993 file transfers were common. At that time File Transfer Protocol (FTP) servers and Usenets (similar to Bulletin Board Systems (BBS)) were the standard way of file sharing. Sharing files through centralized systems is still conventional nowadays, through file sharing sites like RapidShare or Mediafire, and cloud computing services like Google Drive or Dropbox. The use of a centralized client-server model is widely used for file sharing despite all the innovation is decentralized solutions.

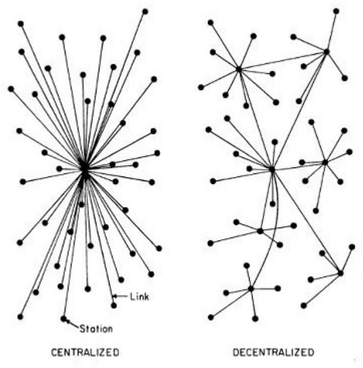


Figure 1.1: Centralized vs. Decentralized.

As shown in 1.1, centralized solutions are very different from decentralized ones at a structural level. In general the advantages of centralized approaches are the disadvantages of decentralized ones and vice versa. The advantages of centralized approaches rely on unified control, easy update and transformation of data, efficient use of resources and an easy structure to maintain and scale. While a decentralized approach provides more redundancy, lower costs, flexible structure, modular upgrade and high effectiveness [8, 25, 38]. The difference between centralized and decentralized approaches can be characterized as efficiency against effectiveness respectively.

Regarding file sharing specifically, the decade of 1990s was dominated by the client-server model, but new decentralized solutions were developed by the end of the decade that became known as Peer-to-Peer (P2P) [38, 39].

1.1 Evolution of Peer-to-Peer

In 1999 a pioneer service was born that started a wave of peer-to-peer (P2P) applications and services for consumers, its name was Napster. Napster was born in July 1999 as a file sharing service focused on music. The program was the first of its kind to allow P2P file transfers using central servers for indexing and search functionalities. Despite its big success in a short amount of time, several lawsuits were filled against the company regarding copyright infringement under the US Digital Millennium Copyright Act (DMCA), law that was passed one year early. Eventually 2 years after the lawsuits Napster filed for bankruptcy. Napster in legal terms had a problem, it's centralized servers were a point of failure and the court stated that if they could detect and avoid the sharing of copyrighted contented they would be obliged by law to do so, in court Napster stated that they had a solution that would remove 99.4% of all copyrighted content, but the idea was regarded as "not enough" [54]. Meanwhile enthusiasts tried to continue what Napster had started, as an open-source project named OpenNap that extended the original protocol, however it had the same exact flaw as Napster, it had central servers that were easy targets for DMCAs. Napster's death started a wave of new programs and protocols trying to take it's place, which made 2001 the year of birth of new file sharing projects, that later became known as the 2nd Generation of P2P [48].

One of those programs was KaZaA, developed by Sharman Networks. KaZaA used a proprietary protocol called FastTrack that was created by the same company, this protocol was later was adopted by other programs [30]. FastTrack was more decentralized and relied on supernodes that were selected depending on fast network connections, high bandwidth and quick processing capabilities. Those supernodes were responsible for handling the search for content and coordination between supernodes.

FastTrack's functionality is similar to Napster's, it's main differences lie on using supernodes instead of centralized servers and optimizing the search for content through it's network of supernodes. However, the developers wanted control on who used the network, so they required that all nodes would have to register on a central server, removing the decentralized aspect of FastTrack's design and introducing a point of failure. [7]

Another of the protocols developed in 2001 was GNUtella. GNUtella allowed a completely decentralized approach where all nodes are equal and where there is no central entity or central server. As a decentralized solution, GNUtella needs to be pointed to previously known nodes for the bootstrap, this could be done by providing a public list of nodes or scanning for nearby LAN nodes, the bootstrap process is slow and could lead to network fragmentation, but this happens by design when using a fully decentralized system. [39]

Comparing the two, FastTrack was a semi-centered protocol and GNUtella was decentralized. Both used a similar way to do searches, while GNUtella would query all the connected peers and those would do the query for their connected peers until this reached 7 hops, on FastTrack a peer would query a supernode and the query would be broadcasted to all supernodes and then to their connected peers for 7 hops. Hops represent the number of times the message has been forwarded, in this case the query would cease to be forwarded after passing through 7 nodes.

Overall FastTrack produced better results querying more than 10 times more nodes than GNUtella, this came with the trade-off of being more aggressive to the network and handling all the data, which was considered as acceptable given how the backbone of supernodes created among normal users [7].

FastTrack's backbone for routing was more stable than GNUtella's, given that supernodes were less prone to disconnecting and were redundant. This meant that on FastTrack, network changes typically would not have a negative effect, while on GNUtella, losing a node would result in being disconnected from a chain of nodes and previous search requests would be discarded.

While the search between these two protocols were a bit different by design, the file transfers were almost identical, both were over standard HTTP [7].

In Distributed Systems the shifts between centralized and decentralized solutions is quite usual [18] [25], and in file sharing this is no exception. Both models have their trade-offs but the evolution of decentralized approaches have been more significant for file sharing solutions. After the first generation (Napster) and the second generation (GNUtella, FastTrack) some problems still remained to be solved [50], network fragmentation, loss of search replies, weak solutions for downloads from multiple sources, no integrity or authenticity verification and finally scalability issues. These problems were mostly solved in the third generation with the BitTorrent Protocol. BitTorrent is a hybrid solution that relies on centralized trackers and in the usage of Distributed Hast Tables (DHT). Stripping itself of the need for a search system, the BitTorrent protocol aggregates users in swarms, that group users by the content they are downloading/uploading, solving network fragmentation problems and content pollution[41].

1.2 Removal of Content

The consumption of content has always been tied to regulatory problems, more noticeably copyright regulation. Centralized solutions have control over the content being served as well as who is consuming it, making it possible to comply with copyright law and its complex rules. These solutions are able to limit the access to the content effectively. Solutions that rely on user uploaded content, like Youtube, are also able to remove content at will. In spite of this, these platforms tend to side with entities that report content for removal, which incentivizes the abuse of the law and shifts the neutral presumption of fair use against the uploader [40, 11, 49].

The removal of content from P2P Networks is a lot more complicated due to its decentralization and lack of central control. The evolution of P2P systems was heavily influenced by these factors, to the point that in BitTorrent there is no way to remove content from the network. This left three options for entities that want to remove content from the network. The first was to target torrent search engines, as BitTorrent has no search mechanism, content is usually found using third-party websites. The second was to target trackers, being the only component in BitTorrent protocol that is centralized, by targeting trackers the users would have to fully rely on the DHT. The third was to target individual peers, by scanning the DHT and deanonymizing peers connected to a specific swarm [55, 9].

For both centralized or distributed solutions, the way the platform deals with the removal of content is an important aspect that determines its success. A level of balance needs to be achieved between complying with all regulations and the system being abused. Security-wise the platform also needs to be able to protect itself and its users, which is a challenge for distributed platforms as external entities attack the network as a mean to remove content. [56, 46]

The United Nations has also been advocating against Internet censorship, which includes protection to consumers of their rights to free expression, fair use and privacy. [35, 34] Content removal is highly connected to censorship as a mean to surppress people and their ideas. While content platforms have different weaknesses regarding the way content is removed from the network, P2P solutions are believed to have the upper hand for defending these human rights. [47, 23]

1.3 Motivation and Goals

The need for multimedia content has been growing these last years [45, 44] and that stimulated the start of several platforms of content distribution like Apple Store, Google Play, Youtube, Netflix, Hulu, etc. Content is distributed in a centralized way and carries high infrastructure costs to the entity that delivers them. However, P2P networks allow the distribution of content in a decentralized way with low costs. These networks require the use of specific applications and technical knowledge of how to properly use them, that turns it hard for the general user to consume the content delivered on these platforms. Due to the inclusion of new features in HTML in its 5th revision (HTML5), an opportunity has arisen to explore the added functionalities, which allow the creation of P2P networks and streaming of multimedia content directly in the browser.

The objective of this dissertation is to take advantage of these technologies that are still in a development phase and that will be integrated into the HTML5 standard, specifically related to WebRTC, P2P and WebSockets to develop a P2P based multimedia file sharing solution. The dissertation makes use of the several technologies that compose HTML5, explores the inner-workings of the BitTorrent protocol and the development of a prototype that allows the distribution of multimedia content using strictly a browser.

This dissertation addresses the way multimedia content is distributed, aiming for a decentralized solution with low costs of operation. To achieve such goal, the solution should connect both to a BitTorrent network and to a WebRTC network in order to share multimedia content between clients. To increase the ease of usage of this P2P application, the solution should run strickly on a browser.

1.4 Document outline

The remainder of this document is organized as follows. Chapter 2 introduces the State of the Art, it is divided in 2 sub-chapters, the first one focus on products and services that are related to this dissertation objectives and the second overviews protocols that are used to achieve it. A solution is proposed in chapter 3, presenting its specification and architecture, explaining why some design choices were made and how it complies with the dissertation's objectives. Chapter 4 provides in-depth information about the solution that was developed. Chapter 5 explores the results of an online survey. The goal of this survey was to determine how users consume content online and what they value more on the platforms they use. Furthermore, test results about the solution are presented. Chapter 6 focuses on main conclusions from this dissertation and some suggestions about future work that could be done if the prototype would to be used on a production setting.

Chapter 2

State of the Art

Distribution of content can be achieved through multiple means. A transfer is the action of transmitting data over a network. To transfer a file a centralized solution can be used, relying on HTTP or FTP, it can be distributed, using P2P protocols like BitTorrent or Gnutella, and it can also use instant messaging services. This chapter is divided in 3 parts, Web Technologies, Chat Solutions and File Transfers. These 3 parts describe multiple protocols and services that use them as a way to distribute content.

2.1 Web Technologies

Web technology is described as the use of mechanics that make it possible for devices to communicate and share resources on the web. While web technologies is often regarded as consumed in a web browser and related to HTTP, it includes not only client-side technologies like mark-up languages (HTML, CSS, XML) and scripting languages (Javascript, WebGL, Ajax), but also server-side technologies and frameworks like PHP, ASP, Web Services, Grunt and Spring, and some data indexing and storing technologies like SQL and NoSQL. Web technologies is a broad term that describes an extensive range of protocols, programming languages and frameworks. This section outlines two different protocols WebSockets and WebRTC.

WebSockets

Web applications and a web sites have been evolving over time, not only the technology but also how the users use them. As browsers became more widely used, web applications became more interactive. Interactivity requires bidirectional communications between the client and the server. This was usually achieved by overusing HTTP to poll the server for updates while doing upstream notifications as normal HTTP calls. This caused high overhead and a high number of connections per client, wasting valuable resources by trying to use

HTTP for something it wasn't designed to. From this need for real-time and bidirectional communications WebSockets was born.

WebSockets was created after some discussions over a TCP-based socket API specification in HTML5 standard in 2008. The idea was so well received, that by the end of 2009 Chrome supported the protocol and by the end of 2011 the standard was finalized and enabled by default on most browsers.

A WebSocket connection starts from a client, the request is called WebSocket handshake and it is very similar to a regular HTTP request with an Upgrade header, like HTTP it uses the port 80 and the port 443 for WebSocket connections tunneled over Transport Layer Security (TLS). [20]

The protocol is message-oriented. After the handshake is finished and the connection is established, text and binary data can be exchanged bidirectionally and with low latency.

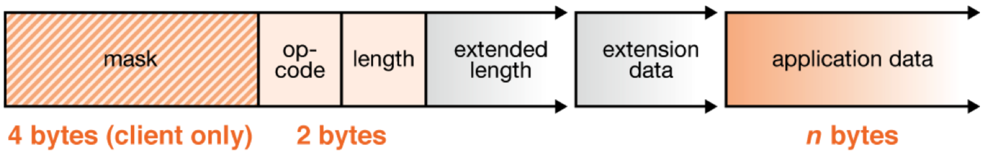


Figure 2.1: WebSocket Frame.

Source: https://www.websocket.org/aboutwebsocket.html

The frames used in WebSockets are really small, which as seen in 2.1 can be from 4 to 12 bytes of overhead per message. Since the protocol also supports subprotocols, it allows both parties to agree on a fixed message format done via JavaScript Object Notation (JSON) encoded messages or a custom binary format. Due to these subprotocols which are specified in message headers, it becomes easier to parse them for both the client and the server.

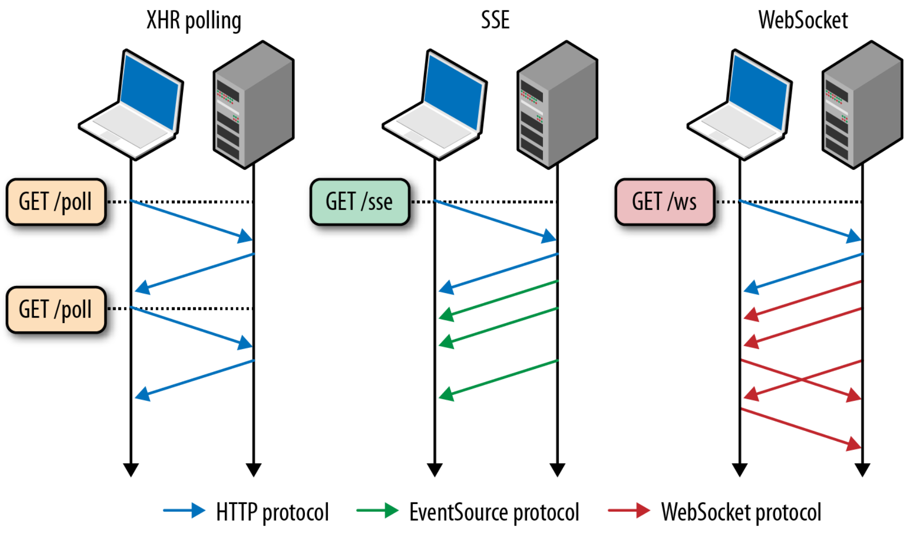


Figure 2.2: Communication flow of XHR, SSE and WebSocket.

Source: https://hpbn.co/websocket/

As seen in 2.1, with a simple comparison between HTTP, Server-Sent Events and WebSockets, it is easy to understand the advantage of both the client and the server being free to send data at any point of time while using WS when compared to the other two alternatives.

WebSocket specification also includes ping pong frames. These frames are used for keep-alive, heart-beats, network status probing and other examples, yet aren't exposed by the API. It is mostly assumed that the server is responsible for requesting pongs whenever appropriate. [20]

Websockets may also be used for broadcasting messages for all clients or sending data from one client to another, but inherently it is a client-server protocol, which makes it less ideal for end-to-end communications between two clients.

Despite using TCP, this API does not allow raw access to the underlying network. Several security measures have been taken so that a multitude of errors are general enough, so that scripts using WebSockets can not be used to scan local networks and expose users information. [20]

WebRTC

Browsers have come to replace a lot of common applications as web applications. Those started to tackle all kinds of functionality inside the browser and making them widely available across different platforms. However, there was a few exceptions of functionalities that browsers didn't seem to be able to deal with, one of those was the voice and video calling capability. After the acquisition of On2 and GIPS by Google in 2010, Google was at a place where they could open source royalty-free replacements as well as a media framework to go with it. Instead of doing simply that, they decided to add a javascript integration layer for browsers and push it as a standard at Internet Engineering Task Force (IETF) and World Wide Web Consortium (W3C), which signaled the creation of Web Real-Time Communication (WebRTC). [29] In May 2011 the project was open-sourced [10] and picked up by several browsers, most noticeably by Firefox and Chrome. In a nutshell WebRTC is a plugin-free solution for audio and video calls in browsers, this allows multimedia communication between different endpoints without any third-party while maintaining a high security level on all data.

The architecture of WebRTC is shown in 2.3. WebRTC exposes 3 core components, the transport layer, the video and audio engines and the JS API. The transport layer handles real time communication and session handling. Video and audio engines have all codecs and hardware optimizations, including echo cancellation, noise reduction, image enhancements, low latency encoding and so on. And lastly, the high level JS API simplifies the use of the other components. As shown in 2.3, the browsers override the PeerConnection API, and are responsible for the lower level Network I/O and acquiring Audio and Video sources, additionally web developers have the WebRTC API that can be used to access all these functionalities using HTML5 and JS, while ignoring hardware or codec specific details.

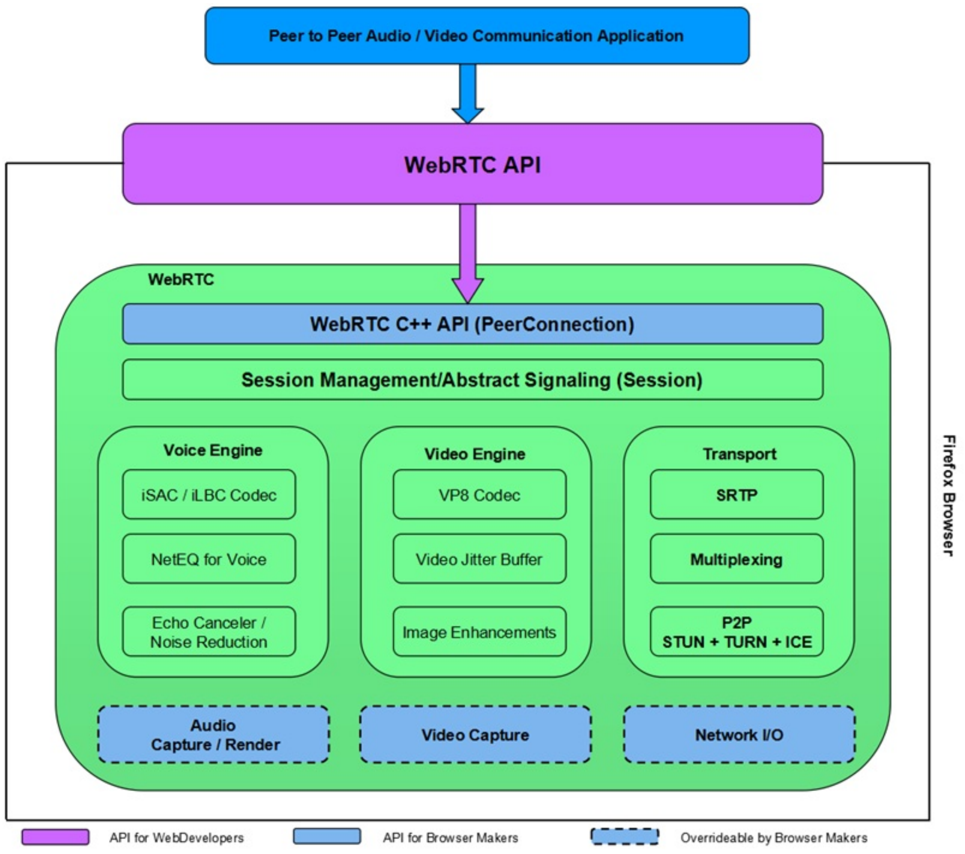


Figure 2.3: Architecture of WebRTC.

Source: http://www.iwavesystems.com/webrtc-peer-to-peer-imx6

WebRTC API has 3 relevant APIs which are getUserMedia, RTCPeerConnection and RTCDataChannel. The first, getUserMedia, is responsible for synchronized streams of media and is where contraints are defined, including which protocols or codecs will be used. These have to be set before a connection is established. The second, RTCPeerConnection, is the main WebRTC component, it handles stable and efficient communication of streaming media between peers. The third, RTCDataChannel, is responsible for real-time communication of arbitrary data with low latency and high throughput.

It might not seem obvious at first, but establishing a P2P connection between web browsers with bidirectional communication is rather challenging. This happens because most machines aren't assigned a static public IP, typically a device is under NAT and a firewall which means there is no direct way to reach them outside of the network, from the Internet. To solve this, WebRTC can use one of two solutions, a Session Traversal Utilities for NAT (STUN) server or a Traversal Using Relays around NAT (TURN) server. The aim of these servers is to reply to the device exposing their public IP and port, so that a bidirectional communication can be done from both sides of two peers establishing a connection.

The use of a STUN or TURN server is part of a larger process called Signaling. Signaling concerns network discovery, session creation, media metadata and codec capabilities, yet it is not specified by WebRTC and web developers are free to choose what technologies and protocols to be used for the signaling process.

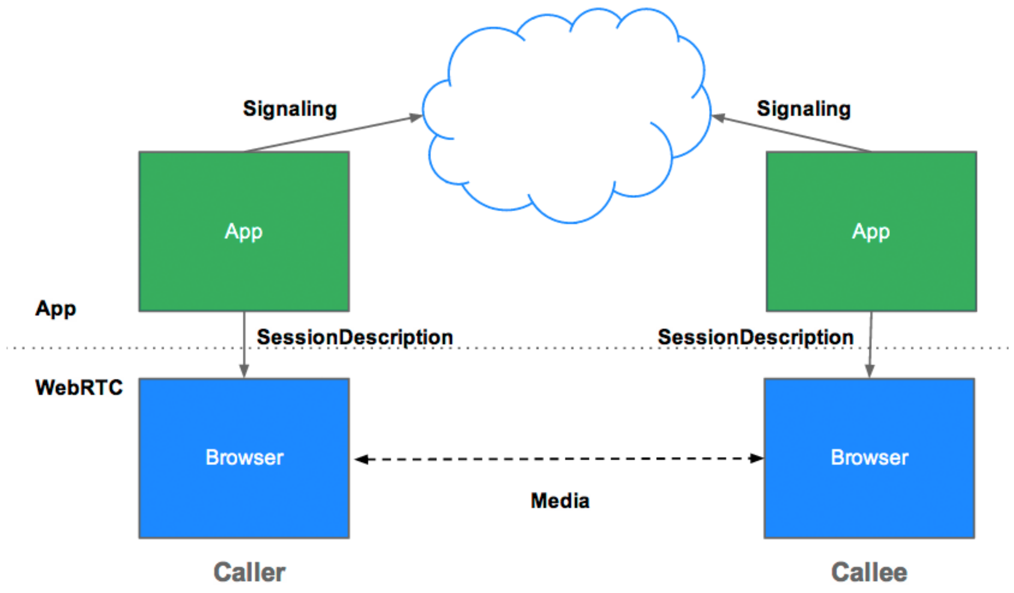


Figure 2.4: WebRTC JSEP Architecture.

Source: http://www.html5rocks.com/en/tutorials/webrtc/basics/

To establish the connection a Session Description Protocol (SDP) needs to be exchanged by both peers, the first one is called the "offer" and the second one is called the "answer", all the media specific metadata is also contained in SDP. This offer/answer architecture is known as JavaScript Session Establishment Protocol (JSEP), as shown in 2.4. After the SDPs are exchanged, both peers generate Interactive Connectivity Establishment (ICE) candidates and send them to each other. ICE candidates are a list of possible IP addresses, ports and protocols to be used by the other peer, after all of them are exchanged, the WebRTC tries to choose the best one to establish the connection.

To establish a WebRTC connection the signaling process has multiple steps that are shown in 2.5. In this example Peer A starts the process by doing requests to the STUN and the TURN server and creating the SDP Offer, which is sent through the Signal Channel. In the example shown in 2.5 the signal channel isn't specified, so it can be any protocol as long as both peers agree on it. After receiving the offer, Peer B creates the SDP Answer and sends it to Peer A through the signal channel. Hereupon both peers start generating and sending ICE candidates to one another, in the example Peer A sends one and Peer B sends after that, however it can happen in multiples ways. Both peers can receive a single list of ICE candidates which each peer will try to pick the best one and establish the WebRTC connection, or both peers can receive multiple ICE candidates, this is known as Trickle ICE, in this case ICE candidates are send incrementally and the WebRTC API will decide when it has a good ICE candidate so it can establish the WebRTC connection. As network discovery

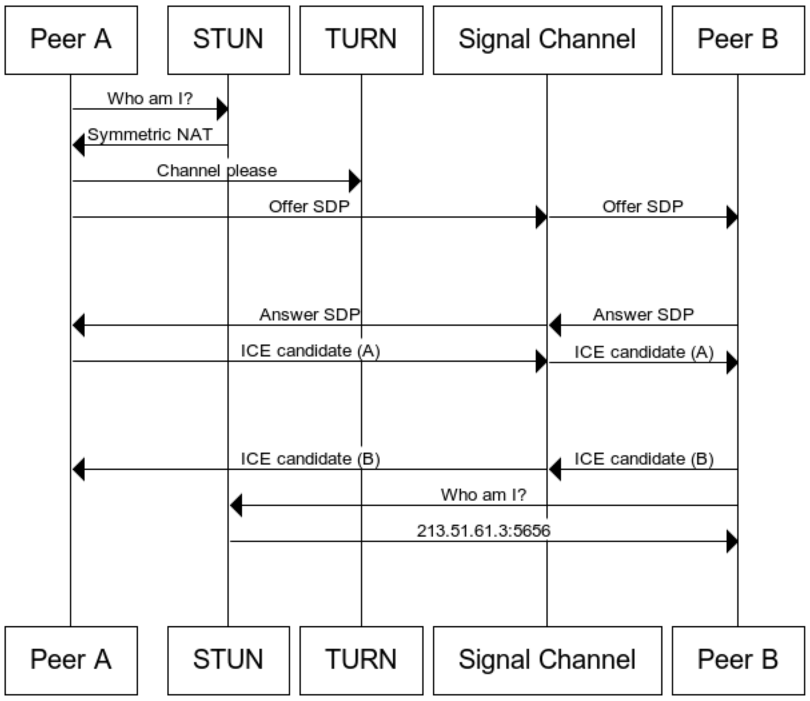


Figure 2.5: WebRTC Connection Diagram.

Source: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity

takes some time, the trickle ICE method usually achieves a much faster connection time.

There are 3 types of ICE candidates: Local, reflexive and relay. Local are actual IP addresses bound to the target device and it is the highest preferred candidate option. Reflexive are candidates with the public IP address given by the STUN request and are the next preferred option. Lastly, relay are candidates with public IP address assigned by the media relay server, that will redirect all traffic to the target device. [27] In simple terms, local candidates are usually chosen when the device has a public IP address that is directly accessible, reflexive candidates are the mostly chosen ones, these rely on using the STUN server to bypass NAT and firewalls, and relay candidates are a fallback, using TURN servers to relay the data through a third-party server when no other option is available.

After the WebRTC connection is established, RTCPeerConnection enables real-time video and audio data to flow between both peers using the setting that were specified when establishing the connection. This is also when the RTCDataChannel can be used to send application data between the two peers, this API is similar to WebSockets API by design, its difference lies in higher performance and lower latency, given that there is no third-party server relaying the data like in WebSockets, but being a P2P and offering customizable delivery properties, as the underlying transport protocols can be chosen.

Encryption is used on all WebRTC components, including the signaling for establishing a new connection. This assures that not only the media streams are encrypted, but also all the setup communication and all further data over data channels. This point is essential as

WebRTC is used for communication between browser endpoints where there is no trusted third-party, so this makes sure that media and data can't be tampered with or eavesdropped en route.

Current implementations of WebRTC have revealed a lot of privacy concerns, these happen because the requests to STUN Servers return a lot of private information, like the public and private IPs exposing that machine and the network. As this information is available to javascript, any JS code can use WebRTC to leak this information to third-parties. This issue became even more serious when it was noticed that the information was leaked even behind a VPN and that these requests to STUN Servers were neither visible in the developer console nor being blocked by the typical privacy plugins like AdBlock Plus, Disconnect or Ghostery. Until the time that document was written, there is still no way to solve this problem except to disable WebRTC entirely, all browsers that support WebRTC leak the information, even in privacy (incognito) mode. [1]

Sharefest

Sharefest[2] project started on a "hackathon", and the objective was simple, a serverless way of sharing files between browsers, though the users still had to go to a URL to download the file, no content was served from the server. The project would do the WebRTC signaling and the peers would share the content with one another. As long as that page was open, the files would be shared to whoever peer that joined the same page. To share content, a initial file would have to be given, then it would create its hash and add it to the URL. As the hashes are unique, any visitor of the said URL would download the file from the remaining peers.

The concept of the project was simple but effective. The download speed was very good and establishing the WebRTC connections didn't take long compared to other WebRTC projects at the time. Sharefest only handles single files, meaning the selected file would be hashed and be accessible by its unique URL. Sharefest had a limitation that caused network fragmentation, due to the implementation of WebRTC being different in both supporting browsers (Firefox and Chrome), this meant that peers would only be able to connect to each other if they were using the same web browser.

By mid-2014 the project was abandoned, and the browser updates broke most of what was working. Some recent tests on different browsers had several unexpected behaviors, but none of the tests could successfully download a file from another peer.

Swarmify

Swarmify[3] is a service similar to a content delivery network (CDN) solution, but instead of using localized servers chosen by geographical proximity, it uses a structure of multiple

[1]https://github.com/diafygi/webrtc-ips
[2]https://www.sharefest.me/
[3]https://swarmify.com/

distributed servers and peers. In essence the service exchanges efficiency for effectiveness, by reducing the load on central servers and CDNs and sharing those load requirements with regular users that are consuming the same content.

From the perspective of a client the solution is simple, swarmify only requires the client to add some javascript code to the site and it manages itself. After that the site will function normally and swarmify determines how viewers of the site share content of the site automatically using WebRTC Data Channels. When a regular visitor accesses a website that uses Swarmify's service, it connects to a WebSocket server and checks if there are nearby peers. If there are, a WebRTC connection is established and they download static content of that page from them, if not, the static content will be served from the website normally. The selling point of this service is that visitors of the site help serving the site's static content like a paid CDN would, however, this resources come as "free" to the website's owner. Swarmify also focuses on video streaming, where parts of the stream might be served through data channels from other peers, effectively reducing buffering time and bandwidth.

Swarmify works as a CDN augmentation service, as it is used in parallel to a CDN, effectively offloading part of the bandwidth to viewers of the site. The service manages to save costs for the site owner and potentially reduce latency.

As Swarmify uses WebRTC, it is limited to browsers that support it, and on those that do support it the same limitation referenced on 2.1 applies, where peers can only share content between other peers using the same web browser. Swarmify doesn't tackle the issue where regular viewers of the site might not want to share their resources, specially for users with monthly data caps. Disabling WebRTC would be an option, but users would also lose all the optimization that comes from the use of Swarmify.

Peer5

Peer5[4] is a serverless CDN solution created by the same developers of 2.1.

It can be compared to PeerCDN[5] and Swarmify, it does similar functionalities but better. Peer5 doesn't try to be a CDN, but reduce the costs on 2 assets specifically, video streams and downloads. Technologically they are more or less identical, but they assure a better service for their peers or else, it defaults to the normal HTTP usage. It is easier to understand for streaming services, normally a user on the site would get the stream for the server or a CDN service, what Peer5 does is try to find if any peer on the network that is watching the same stream, is near than the CDN/Server, if it is, it uses it instead, assuring a better service for the user and a bandwidth save from the server perspective. Also, if the peers are mobile devices they won't ever try to serve content to the network, as they consider it too unreliable.

Peer5 can never deteriorate user experience and it does real-time switching between P2P and the Server as needed, this also means that the stream will have less delay than the original solution (without Peer5). It has some disadvantages though, users on non-compatible

[4]https://www.peer5.com/

[5]https://github.com/PeerCDN

WebRTC browser will use the server normally and Peer5 will be ignored if there is no peer that can give a better service than the server, this can be especially hard if the content isn't popular, however, popular streams are usually the bottleneck on CDNs, so it does complement that CDN's weakness pretty well.

As specified on their FAQ, they don't use BitTorrent in any way, they have a proprietary stack designed for WebRTC and hybrid rich content delivery. That is their selling point, but their solution is very convenient for usual downloads as they connect every user downloading the same content and they share it between them as long as that page remains open. It is not specified if the high reliability requirement on the peers stated for streams is used for this type of content, as the reliability regarding delay isn't as important for static content.

2.2 Chat Solutions

Instant Messaging (IM) has a long history and always existed in different forms since computers were connected to networks. Instant messaging is usually defined as text-based communication between two or more participants over the Internet, allowing effective communication between recipients that are online. It differs from other solutions like e-mail because of it's real-time component. IM applications grew to have a wide range of functionalities, some more typical like chat rooms, file transfers and profiles, others even having voice and video chatting built into them. While most of the IM solutions rely in a centralized model, some functionalities like file transfers or video chatting are done directly between the user clients. Of all the chat solutions in the market two protocols will be explored in more detail, IRC and XMPP.

IRC

Internet Relay Chat (IRC) was born in 1988 and was created with the intent to extend the BBS software. IRC is an application layer protocol over TCP that uses the client-server model, where every message has to be relayed from the server to the clients.

An IRC server can connect to other IRC servers to expand the IRC network. Users that are connected to one of these servers can communicate with any users seamlessly indifferent to what server they are connected to, as long as the servers are connected to each other. The structure of an IRC network can be perceived as a tree, messages are routed like multicast, where the same message travels only once per network link. IRC Servers only relay messages that need to be consumed on other servers or clients connected to them, the only exception are network state messages that are always relayed to every server.

While IRC Networks are not as widely used today, most of the chat concepts it popularized are still in use by most real-time chat applications, like channels or chat applications being command-oriented. Channels can be understood as groups of users, where the group is defined by a name that starts with a number sign "#". Channels be joined or left by users and have their own set of rules. IRC also establishes a hierarchy of powers inside a channel, where users might have different rights that affect the channel and users inside of it.

XMPP

Extensible Messaging and Presence Protocol (XMPP) is created as an open-source protocol that seeks to standardize the usage of IM across the market. There's several types of IM services, but most of them are very different and rely in proprietary code. These turned out to be incompatible between with one another, despite serving very similar core functionalities. XMPP came to solve that as an Internet Engineering Task Force (IETF) open standard, using XML data format and allowing simple extensions to the core protocol.

As an open standard XMPP had improved rapidly and led to its wide adoption across different vendors. Its adoption by popular IM clients like Google Chat and Apple's iChat brought compatibility among them, allowing users to chat with each other even when using different applications. XMPP uses Extensible Markup Language (XLM) "stanzas" for protocol communication, which essentially is a fragment of XML that is sent over a stream. This way of structuring data let's XMPP to be dynamic and contain more data depending on the extensions and functionalities the server has. Through XML namespaces it is easy to transport custom data in addiction to standard messages without affecting clients or servers that don't support the extra functionalities. Using XML is also an advantage, given its popularity as a data exchange format that is a standard for a lot of software, which simplifies integration with other existing solutions.

XMPP network is formed by all XMPP clients and servers that can reach each other. While it is possible to create a private XMPP network within an internal LAN, public servers on the otherhand are connected to each other in a big network.

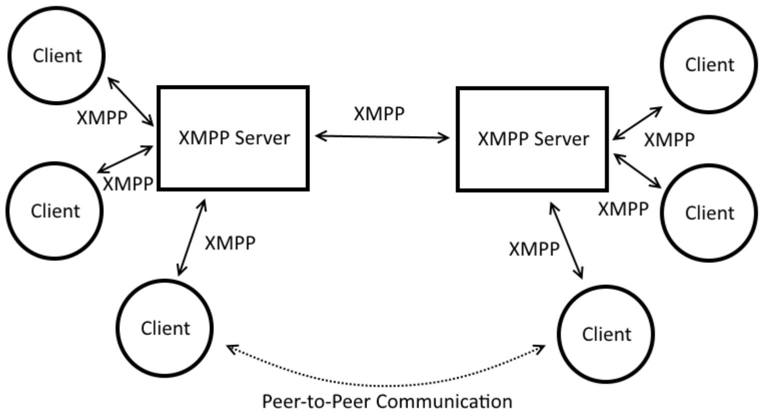


Figure 2.6: XMPP Client and Server Communication

XMPP uses globally unique addresses based on the Domain Name System (DNS). This enables all XMPP entities to be addressable to one another and deliver messages over the network. Similar to e-mail, it has the user followed by the domain name, for example "user@example.com". As seen in 2.6 clients connect to their server, meanwhile the servers connect to other servers for communication with external clients. As DNS is used, locating the target server is trivial, turning the routing process easy to handle for server to server communication. XMPP addresses are generated and authenticated by the owner of the domain in question, hence the server is responsible for its user accounts. XMPP uses a client-server architecture where the server is responsible for handling most of the complexity, these include the authentication, message delivery and the presence information of all users within the domain. This allows the XMPP clients to be simple and lightweight. For communication with external users a server-to-server model applies, which means the server delivers the message to the responsible external server and it is therefore responsible to deliver it to the assigned user within it's domain. This way of dividing responsibilities keeps cross-domain communication scalable but also flexible enough to meet the requirements of individual domains.[43]

XMPP uses long-lived TCP connections in both client-server and server-to-server communications. This has the advantage of bi-directional communication, allowing the server to push data to a client whenever needed but has the disadvantage of lower reliability, given that disconnects take longer to detect. All of the communication channels have robust security using Transport Layer Security (TLS), while end-to-end encryption is possible to be achieved by the use of extensions. [6]

Some extensions of XMPP also allow peer-to-peer communication between clients (as shown in 2.6), this is typically used for direct audio and video streams as a P2P connection provides better latency and lower server costs.

2.3 File Transfers

Peer-to-Peer applications are one of the major ways used today for file transfers over the Internet. [44, 45] Across the three generations of P2P, the evolution of P2P solutions have matured the protocols that are currently in use and have pinpointed the characteristics that have determined their success. Despite 3rd generation protocols all having some kind of hybrid compromise, these protocols have structured themselves distinctly. Two protocols will be explored in this section, BitTorrent and Gnutella, additionally, the BitTorrent section explores some applications that use the protocol to accomplish unique results.

BitTorrent

The BitTorrent protocol is a peer-to-peer file transfer protocol where large amounts of data can be transferred between users, and it employs several mechanisms to encourage good behavior on the network without enforcing it with any central entity. In the early implemen-

[6]https://xmpp.org/rfcs/rfc3923.html

tation of the BitTorrent protocol it relied on a centralized tracker, a torrent file and a swarm of peers.

For BitTorrent all the content is divided in pieces of equal and specified size (by the torrent file), this happens even if multiple files exist in the content. A piece can contain, for example, multiple small files, or the end of a file and the start of another, the BT Client only needs to assemble the files correctly when the piece is written to the disk, in transit the client only has to handle pieces. This makes the transfer easy and standard regardless of the number of files in the content. Every piece has the exact same size, except the last one which has the remaining data and doesn't fill the entire piece. The size of pieces are a power of 2 and typically grow depending on the total size of the content, therefore the piece lenght of a torrent should be balanced. Whilst opting for an excessive piece lenght would lead to inefficiency, setting it too small would cause large torrent files and more overhead.[3]

BitTorrent's file distribution system starts by loading a torrent file into a BT client, this file is usually downloaded from a web server where some info about the content of the torrent is given. The torrent file has no data, it is just metadata that contains names, folder structure, sizes and hash values. The torrent file's structure uses an encoding algorithm called Bencode, while it isn't a friendly format for reading, it assures uniqueness in a compact binary format. Bencode can hold byte strings, integers, lists and dictionaries. Strings are encoded as a key and value pair, the key has the length of the string as an integer in ASCII and base10 format, while the string is coded in binary format, so a string is encoded as follows "<string length>:<string data>". For the other 3 types, bencode has an initial letter to identify the type, "i", "l" and "d" for integer, list and dictionary respectively, and a tailing "e" independently of the type.

Strings
3:bob represents bob
5:alice represents alice
Integers
i133e represents 133
i0e represents 0
i-50e represents -50
Lists
l3:foo3:bare represents ["foo","bar"]
l3:numi20ee represents ["num": 20]
Dictionaries
d3:foo3:bar3:numi20ee represents {"foo": "bar", "num": 20}
d5:namesl3:bob5:aliceee represents {"names": ["bob", "alice"]}

Listing 1: Bencode examples by data type.

Bencode allows complex structures as Lists and Dictionaries can contain all the other 4 types and the algorithm doesn't specify any restrictions for these types. Despite not being as human-readable as JSON or XML, it's encoding and decoding is fast, holds binary data while maintaining an acceptable low size and has no limits for integers or string lengths. [5]

The torrent's structure is rather simple and is represented as follows:

```
- Announce
- Info
– Name
– Piece Length
– Pieces
– Length
– Files
— Path
— Length
```

Listing 2: Structure of torrent metadata.

The Announce contains a tracker or a list of trackers, the Info is a dictionary of unique keys that contains all the metadata of the content. Inside Info there's the Name that contains the filename if the torrent has a single file, or the directory name if it contains multiple files inside it. Then there's Pieces and Piece Length, the Piece Length specifies the number of bytes in each piece and the Pieces contain the hash of every piece by order, this is used by the client to perform integrity checks and validate if the piece received is correct. The Length specifies the size of the file in bytes, if the torrent has only a single file it is under Info, if not, every object in Files will have a Length. Files only exists if the torrent has multiple files, in this it will have an object for every file, each one has its Length and Path, the Path has the filename but can also specify several directories under the root folder of the torrent (that is on the Name) where the file will be located. The files are ordered, and the hashes in the Pieces respect the order of the listed files, this is important because a single piece can have several parts of different files. The torrent can also have other optional information, like Comment, Created By, Creation Date, Private and much more. The Private is a boolean that informs the client whether it should stick to the swarm given by the tracker and disable other peer discovery methods. As the BitTorrent protocol evolved, the information stored inside torrent files suffered some changes. BitTorrent clients can opt whether they use the extra information and whether the client has needed capabilities to use them (e.g. for new features).

As an example the information contained on a torrent file is presented below for a torrent for a Linux Distro (Kali Linux[7]):

[7]https://www.kali.org/downloads/

```
{
"announce": "http://tracker.kali.org:6969/announce",

"announce-list": [ [ "http://tracker.kali.org:6969/announce"], [
"udp://tracker.kali.org:6969/announce"] ],

"comment": "kali-linux-2016.1-amd64",
"created by": "ruTorrent (PHP Class - Adrien Gibrat)",
"creation date": "1453327437",

"info": { "files": [ { "length": "2945482752", "path": [ "kali-linux-2016.1-amd64.iso" ] }, {
"length": "70", "path": [ "kali-linux-2016.1-amd64.txt.sha1sum" ] } ], "name":
"kali-linux-2016.1-amd64", "piece length": "262144", "pieces": <Binary Hashes> }

}
```

Listing 3: Example of a torrent metadata.

Pieces information were removed for readability and the Bencode was converted to JSON for the same reason. The converter used for this example is present in the footnote.[8]

For the BitTorrent client to download the torrent it has to connect to peers that have the content, to do this the BitTorrent Protocol uses a Tracker. A tracker is a HTTP Server that typically responds only to 2 URLs, the announce and the scrapper. The Announce receives the parameters from the BitTorrent client and responds with a list of peers of the requested torrent while the Scrape is a subset of the announce, and only gives info about the number of peers and seeds. Scrape was used in the early implementations of the BitTorrent protocol because request were light and the client could make decisions based on it, for example, requesting for more peers earlier than expected when the reported pool of peers/seeds is high. Scrape was also used in the early days to check tracker data, if the request came from a browser it would report a more user-friendly page with the same information: torrent's name, number of total downloads, current peers and seeds. [1]

BitTorrent was built as a hybrid solution, the tracker is the only centralized part of this protocol, as clients connect to the tracker they inform the tracker of their current progress on the torrent and it keeps information about it. This became useful for private trackers that monitor their users, implement rules on download/upload ratio and have control whether the users can continue to use the tracker or not. The protocol wasn't made with this in mind, the word "private" might be a bit deceiving here, as the communication between peers has nothing private about it, if other discovery methods are used the swarms of peers are rather public even if the private tracker will only keep track of the clients directly connected to it.

[8]http://marquisdegeek.com/code_bencode.php

The BT Client's request to the tracker (called get-announce from now on) contains info on the state of the torrent (status, size in bytes downloaded, uploaded and left until 100%) and information of the client (IP, port and peerID) and finally the info_hash of the torrent. With this info the tracker adds the client to the peer list and saves the info that it finds relevant. Get-announce also has 4 possible events, started is when a client starts a torrent and the tracker adds the client to the peer list, stopped is when the client stops the torrent and it is removed from the peer list, completed is when a client finishes downloading the content and turns from a "leecher" into a "seeder" and empty when no event is specified, this is used when a periodic get-announce is made to the tracker. The purpose of events is to keep the peer list fresh and updated, making it possible for the tracker to keep track of seeders, leechers and how many peers finished the torrent. After the client's get-announce the tracker always replies with a response-peerlist, if successful this response contains the total number of leechers and seeders and a peer list. Unless the torrent has a low number of peers, usually this peer list doesn't contain all the peers that are in the swarm. Trackers have this number picked and only return a random number of peers for the peer list. In the early days the default was 50, nowadays as clients don't rely only on trackers, the default on most trackers is 20. [41] This is an important aspect of the tracker, a client doesn't need to connect to that many peers to download, giving the full peer list would be create an unneeded load for the tracker and all the peers in the swarm, without much positive return out of it. Due to this limit some clients would "hammer" the tracker doing multiple requests so they could get a bigger pool of peer to select from, most trackers deal with this by temporarily banning the client's IP which made ill-intended clients drop this and try different approaches.

As BT Protocol became more widely used, the trackers were a usual bottleneck, despite some efforts to encourage clients to query trackers as less as possible and decrease query frequency, this wasn't enough. As HTTP trackers had significant overhead, in 2008 UDP Trackers became a standard, effectively reducing the traffic by 50% and reducing CPU usage of the trackers, the new UDP implementation made trackers more light because complex parsing was no longer required as well as no connection handling. [22] The protocol specification for the UDP Tracker almost identical to the HTTP specification [22, 51]. However, due to UDP connections being easy to spoof the tracker could be used in DRDoS attacks, to prevent this the first 2 packets exchanged set a connection ID that ensures it doesn't happen, this can be considered a simpler TCP handshake. [22] This connection ID is like a session cookie in HTTP approaches, it has a 2 minute timeout and can be used in multiple get-announce requests, if the connection ID is invalid the packets are dropped and if it expires a new one has to be requested by the client. It is a good policy to limit the allowed request number of peers from the tracker side to keep the size of the requests small and also avoid packets higher than 1500 bytes to avoid transport layer packet splits. As most of BitTorrent's implementation it allows the specification to be extended without breaking compatibility.

These concepts just mentioned are intertwined on the peers side, as the clients are decentralized and totally independent, some software clients might even choose to use the protocol in different ways. To keep it simple the fig. 2.7 shows generically how a client bootstraps and is able to join its intended swarm of peers for a given torrent file:

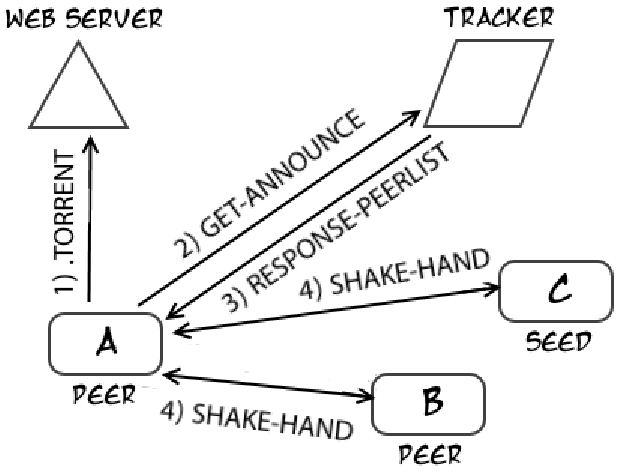


Figure 2.7: BitTorrent Client joining the swarm.

The client first has to load a torrent file (step 1), as stated previously this file has all the metadata needed to checksum pieces, create the empty shells of all files and information on how to join the swarm, usually given by the tracker. This step is done by the user, that picks a torrent file from a third-party and loads it into a BitTorrent client.

The client requests a get-announce from all the trackers that are listed in the torrent file (step 2), by doing these requests the trackers will add the client's info to the pool of peers for that torrent. Then the client waits for successful response-peerlists (step 3), these lists contain only a small number of peers which the client picks some of them and tries to connect to them.

The next step is to shake-hand with the selected peers (step 4), the packet is simple, it identifies the BitTorrent Protocol and contains the hash of the torrent and the peerID. If both peers do this successfully the connection is established and data can flow in either direction.

After the client is connected to several peers, it is now part of the swarm of peers of this specific torrent. Assuming the peers A successfully established a connection to the peer B and the seeder C they can now trade pieces like shown in the fig. 2.8:

In this case on Figure 2.8, the peer B is a leecher, meaning it didn't finish the torrent yet, and C is a seeder, meaning it has already completed the torrent and it only needs to upload the content to other peers. In this example A just started this torrent so it has no content to trade with B, so it should request pieces to C that B doesn't have, so it can trade and optimize download speed, more about this will be explored below.

After the client joins the swarm of the torrent, there are only 9 types of messages peers can exchange: Choke, unchoke, interested, not interested, have, bitfield, request, piece and

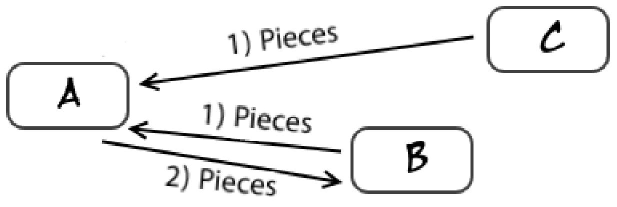


Figure 2.8: BitTorrent Client downloading pieces.

cancel. [22] This excludes handshakes and keepalives, that have an empty message type and no more fields except those in the header (fields for protocol, peerID and torrent hash).

The first 4 have no payload and just define a state to the peer it is sent to. Have message has a index of a piece that has just been downloaded successfully. Bitfield message is only sent after handshake and contains a map of all pieces that the client has, this exposes all pieces that the client has and which ones it is missing. Request message has the index of the piece, a begin offset and a length specifying the length of the request, this message is used to request sub-pieces. Piece message has the index of the piece, a begin offset and the piece data, this is not necessarily a full piece as they can come as several sub-pieces. The cancel message contains the same index of a piece, an offset and a length, and is used to cancel previous requests, cancel messages are usually used in endgame mode.

Another extra type of message in nowadays bittorrent clients is the message port, this message specifies the port used for DHT functionality.

For the file transfer itself the torrent data is divided in pieces and each piece is divided in blocks (also called sub-pieces), traded using tit-for-tat strategy. Tit-for-tat, in short meaning “this for that”, putting it simple terms it cooperates with the other party if they cooperate back, this concept gets the best results in the prisoner’s dilemma [19] and achieves the best resources utilization when compared to any other cooperative techniques known today [14]. In BitTorrent specifically tit-for-tat means that peers will reciprocate uploading to other peers that upload to them. This concept also applies to connections, to be effective the peers have a limited number slots for peer connections, meaning that both peers have to benefit from that said connection, otherwise the peer might choose to connect to another peer that gives a better payoff to the use of that connection slot. [**?**]

There are 2 states that are used to regulate how sub-pieces are transferred through a peer connection, choke/unchoke and interested/not interested. Requests and subsequent transfers will only occur if the state is interested and unchoked. When a peer establishes a connection, it starts as not interested and choked, the peer will send interested to another peer if they want pieces that they have, but requests will only be made if the other peer sends an unchoke. These 2 states exist for both directions per connection, meaning that the transfer of sub-pieces might occur in only one direction and not necessarily in both. To stop a transfer only one of these states has to change, this is what the bittorrent protocol uses to regulate transfers speed.

As a connection is established, bitfield messages are exchanged, this informs both peers about what pieces they have. As new pieces are acquired, have messages are sent, so the bitfield is updated. This is crucial to know what pieces other peers have and can be requested, interestingly this also gives the client some idea of the transfers speeds of the peers it is directly connected to, by looking at how frequent it receives have messages. Using this information the client decides what pieces to request from others, this is important for a good performance as the client needs a strategy to get pieces fast, as well as getting "rare" pieces so it can trade later. Algorithms can get complex for piece selection but 4 concepts are essential: Strict Priority, Random First Piece, Rarest First and Endgame Mode.

Strict Priority policy stands for giving priority to sub-pieces of pieces that have already started being downloaded, this policy encourages the client to finish incomplete pieces before starting to request new ones. Random Piece First is a policy that is used when the client has a low amount of pieces, the client requests random pieces in hope of getting pieces quickly. Rarest First is the general policy that is followed, where the client requests the rarest pieces of the peers it is connected to, this policy works well because the client will always have pieces to trade with other peers leaving pieces that are more common for later, the likelihood of having nothing of interest to trade is therefore reduced. Endgame Mode is a policy that is followed when all pieces have been requested and the torrent is close to being finished, as some peers are slow and could delay significantly the ending of the torrent, these last pieces are requested to multiple peers in hope to finish it faster, as sub-pieces arrive the client sends cancel messages to other peers for which the sub-pieces were requested. However it isn't uncommon that the client receives copies of the same sub-piece which reduces the effectiveness of the network as these sub-pieces are discarded and the bandwidth is wasted, which is the price to pay for this policy. [14, 15]

Congestion control has always been a problem to deal with on P2P applications, this is the case because it is hard to control bandwidth over many connections at once, which is always the case in P2P. To help mitigate this problem the BitTorrent protocol has the choking mechanism as a tool to control and ensure a more consistent download rate. There are a lot of choking algorithms but some concepts remain constant. Choking algorithms follow variants of tit-for-tat, meaning that they should reciprocate to peers that are uploading to the client. There should be a fixed number of unchoked peers at a given time for upload (usually 4), these are chosen as efficiently as possible to achieve good return, pieces from them as a trade, generally the choking algorithms selects the peers with more bandwidth. As connections need some time to optimize and stabilize speed-wise, choking and unchoking shouldn't be done in quick succession, this is known as "fibrillation" and should be avoided. Lastly there's optimistic unchoking, this ignores every metric of the peers and uploads for 30 seconds to a random peer, optimistic unchoking has good results because the peer might try to reciprocate in the near future, and as the selection is random it gives a chance to peers that have low amount of pieces. Newly connected peers are also more likely to get selected by optimistic unchoking. [14] [3] Choking algorithms are interesting because they value bandwidth and make decisions based on it, the idea of having a low number of slots also achieves a better use of resources which isn't obvious at first. When the client is a seeder it no longer has download rates to decide which peers to unchoke and it no longer uses a tit-for-tat strategy, to get better value for the client's upload 2 strategies are used, uploading to peers with the best upload rates and uploading to peers that no other peer is uploading to. These gives pieces to peers that have good upload rates and are able to contribute more to the swarm and to peers that aren't getting pieces, giving them the opportunity to have pieces to trade and make better use of their resources. While this strategy for seeders works well, a specific strategy for initial seeders called superseeding achieved better use of resources in that specific case.

When the client is a seeder it no longer has download rates to decide which peers to unchoke and it no longer uses a tit-for-tat strategy, to get better value for the client's upload 2 strategies are used, uploading to peers with the best upload rates and uploading to peers that no other peer is uploading to. These gives pieces to peers that have good upload rates and are able to contribute more to the swarm and to peers that aren't getting pieces, giving them the opportunity to have pieces to trade and make better use of their resources. While this strategy for seeders works well, a specific strategy for initial seeders called superseeding achieved better use of resources in that specific case.

Superseeding is an algorithm designed for optimizing seeding when there's only one original seeder. When a torrent is created and shared usually there's only a single seeder, as the seeder has limited bandwidth it becomes a bottleneck for all peers, even when peers focus on the rarest pieces, the seeder could upload more than 150% of the total size of the torrent without any of the peers becoming a seed. Using Super Seed Mode the original seeder masks itself as a regular leecher with no data and sends them have messages of rare pieces that were never sent before, this lures the peer to request them. As the seeder will only use it's upload to send unique pieces that were never sent before it doesn't waste any bandwidth, achieving greater results than seeding normally, both in higher speed in the swarm network and faster time to get need leechers become seeds. [3]

In 2005 one of the popular BitTorrent clients at the time, Azureus, came up with the first implementation of trackerless torrents. This is achieved using a Distributed Hash Table (DHT), it provides a lookup service to hashes, pairs of keys and values that is distributed among different nodes, inherently DHT implementations are autonomous, fault tolerant and scalable. There is no central coordination for the whole DHT network, a high number nodes are able to join and leave the network without the routing between the nodes hardly changing, making the whole network reliable and scalable. Typically this ends up as a trade off between efficiency and latency. [16] DHTs provide a structured key-based routing attaining decentralization like Freenet and Gnutella with the efficiency and guaranteed results of Napster, however it only supports exact-match search rather than keyword search. [13]

There are several types of DHTs, their characteristics remain similar but they differ in structure, routing algorithms and performance. Azureus's DHT implementation is based on Kademlia, later on that same month BitTorrent, Inc came up with their own DHT implementation also based on Kademlia, known as Mainline DHT.

Kademlia is a DHT that uses UDP to interact with it's wide range of nodes that forms an overlay network. An overlay network is a virtual network formed of virtual links that runs on top of the Internet. Each node is identified by a unique node ID and keeps lists of neighbor nodes with "close" node IDs known as k-buckets, their node ID doesn't only identify a node uniquely, but is also used as a way of routing and key lookups. In Kademlia the distance is calculated as the exclusive or (XOR) of 2 node IDs or a node ID and a key. This results in nodes that are neighbors because of their random node IDs but that can be in widely different geographic locations.

Kademlia prioritizes nodes that have been connected for a long time, this provides a stable network and resilience to attacks, as these valid nodes typically remain connected for a long time in the future.

As stated on chapter 1, for a P2P system to be healthy and grow in number of users, it has to assure data integrity, high availability, good scaling for a high number of users and discourage parasitic behavior. BitTorrent managed to achieve all of these points in an early stage of development, however BT solution lacks certain aspects by design. To use BitTorrent most users rely on other components for several functionalities, the most relevant being search systems, servers for metadata files (.torrent) and moderation systems to ensure data integrity. [41]

On the early days third-parties that were responsible to keep the trackers would also give this service of providing torrent files and some kind of search capability. This became less common as the protocol grew in use and legal problems became more frequent, meaning that search engines started to give exclusively that functionality while using 3rd-party public trackers, and public trackers became mostly donation-based, independent and avoid any data aside from the swarms. The exception to this are private trackers, that usually have their own trackers, provide torrent files as well as search capabilities while controlling the peers behind a login system, relying on exclusivity of content and better performance as its sell point. [32]

Maelstrom

The beta version of this project allows streaming directly of the BitTorrent network into the browser[9]. This browser however has a built in BitTorrent client into a modded version of Chromium, made from the BitTorrent Company itself. The intent of this project is to be able to read websites delivered in torrent form, while completely different from the target of this dissertation it is an interesting project and a good way to use the BitTorrent technology as a way to serve decentralized websites in a free, open and uncensored network.

[9]http://blog.bittorrent.com/tag/maelstrom/

There is also ZeroNet[10], a similar project that uses the exact same concept but uses Bitcoin technology and is open-source.

WebTorrent

WebTorrent[11] is a open-source torrent client with streaming capabilities that uses WebRTC connections to feed content for clients unable to use UDP/TCP connections. As an overview, it is a BitTorrent implementation over WebRTC, the solution is written in NodeJS[12] and made available to the browsers through Browserify[13], making the browser clients connect to the rest of the peers over WebRTC.

WebTorrent's approach tries to replicate the BitTorrent protocol over WebRTC in a way to encourage BitTorrent clients to support WebRTC connections directly, enabling any browser to be directly connected to the BitTorrent clients without any hybrid solution between the WebRTC and BitTorrent network.

WebTorrent solution has two types of peers, hybrid and webtorrent peers, the first is the implementation on NodeJS with a complete BitTorrent client and WebRTC capabilities, the second runs in the browser on Javascript and is connected only to WebRTC peers.

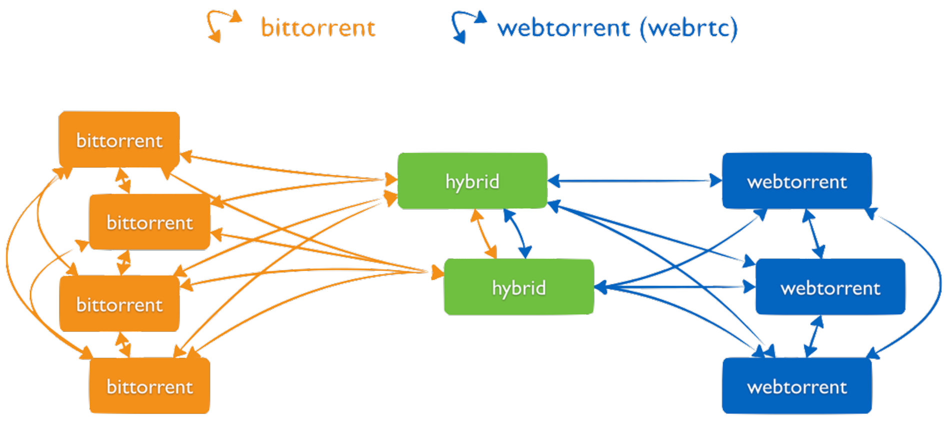


Figure 2.9: How WebTorrent connects to BitTorrent

Source: https://github.com/feross/webtorrent

WebTorrent is divided in 14 NodeJS modules:

Of those 14 module, 2 of them are extensions, non-standard implementations of PEX Protocol and a way to download metadata to support downloads via magnets (without the torrent files). On the plus side, the division on different modules eases the development of the

[10]https://zeronet.io/
[11]https://webtorrent.io/
[12]https://nodejs.org/en/
[13]http://browserify.org/

Module	Description
webtorrent	torrent client
bittorrent-dht	distributed hash table client
bittorrent-peerid	identify client name/version
bittorrent-protocol	bittorrent protocol stream
bittorrent-swarm	bittorrent connection manager
bittorrent-tracker	bittorrent tracker server/client
create-torrent	create .torrent files
ip-set	efficient mutable ip set
load-ip-set	load ip sets from local/network
magnet-uri	parse magnet uris
parse-torrent	parse torrent identifiers
torrent-discovery	find peers via dht and tracker
ut_metadata	metadata for magnet uris (ext)
ut_pex	peer discovery (ext)

Table 2.1: WebTorrent Modules

project and makes it easier for multiple developers to work on the project without breaking it. If some of the modules are good enough they might be used by other related repositories which is one of the purposes of open source projects.

WebTorrent Hybrid Client[14] uses both WebRTC and TCP/UDP, where WebRTC is used for the WebRTC network and the TCP/UDP is used to connect to the BitTorrent network. This app is written in NodeJS, the initial development had planned for WebTorrent's hybrid client to become a chrome extension, but problems with the Chrome API led for it to continue as a NodeJS client. This client works in MacOS and Linux, but because of implementation problems of the WebRTC dependency (wrtc[15]) it has no Windows support. This limitation however is due to the usage of NodeJS libraries that are still under development and which leads to unstable builds in the short term. These limit somewhat the usage of WebTorrent's Hybrid Client by new users, usually leaving it's usage to techsavvy-only users.

WebTorrent project intents to push its BitTorrent implementation over WebRTC and encourage current full-fledged BitTorrent clients to implement WebRTC support using WebTorrent's specification of the protocol. Remarkably the Hybrid Client isn't very important in the long run, as it is a tool for this early stage until BitTorrent Clients are encouraged to implement access to the WebRTC network.

An aspect of WebTorrent project is that it started to implement a BitTorrent Client in NodeJS and then proceeded to convert it to "pure" JavaScript through Browserify. Browserify[16] is a tool for compiling NodeJS code into plain JavaScript that can be run on any browser, this is accomplished by resolving all dependency modules used by the NojeJS application and that are then concatenated into a single self-contained file, browserify handles the conversion

[14]https://github.com/feross/webtorrent-hybrid
[15]https://github.com/js-platform/node-webrtc
[16]Documentation: https://github.com/substack/node-browserify

of NodeJS modules into browser compatible code and manages to create a negligible overhead despite concatenation of all the code. The project's approach have constraints, some modules are simply impossible to browserify, others have certain libraries that have to be overloaded. Ultimately this choice requires a lot of maintenance and a lot of bugfixing as browserified code can misbehave, specially as sometimes parts of the code are completely different from the NodeJS client (hybrid) to the browser client (WebRTC-only).

Most of the current uses of WebTorrent rely on consuming content that is on the BitTorrent network. For this to happen, the solution requires a hybrid client to bridge the two different networks so the content can be consumed. Without a hybrid client for a given magnet or torrent, no content will be downloaded from the BitTorrent Network and clients will have to rely on WebRTC-only peers (also called as "web peers"). While some discussions point to hybrid clients serving as proxies in the meanwhile, or a need for specific BitTorrent to WebRTC bridges, currently such solution has been implemented. Ideally the full-fledged BitTorrent clients will support WebRTC, discarding the need for such bridges or for peers to serve the community from goodwill and bring content to the WebRTC network. At the time writing there has been some development on WebTorrent support as a plugin in Vuze BitTorrent client.[17]

As an open-source project that has its own contributing community, WebTorrent is already used as a library for many other third-party projects and continues to grow in relevancy among P2P solutions . As BitTorrent today dominates the P2P environment and WebTorrent is a key project to push BitTorrent to different clients using WebRTC, some great changes might come from this project and change the way we share data across different devices.

Gnutella

Gnutella was designed to be a P2P system based on an unstructured overlay that allows fully decentralized sharing of files between its peers. Unstructured overlay network stands for the way the overlay is built by establishing random connections to neighboring peers. These have the benefit of having a low cost to build and maintain while keeping a natural level of redundancy and a shared load among all the nodes in the network. [28] This overlay network is used to search for content by flooding, which every peer forwards requests to all its neighbors until a time-to-live (TTL) is attained. Gnutella has no central server, so a new peer needs to know at least one Gnutella peer to join the network. However the protocol doesn't specify any particular way for a peer to bootstrap, as such the multiple possibilities on how to bootstrap remain as a choice for the user.

Gnutella operates in a randomly assigned overlay that is formed of peers that are considered equal in all regards, and the protocol defines 5 types of messages they can exchange:

Ping - Probes other peers

[17]https://wiki.vuze.com/w/WebTorrent

Pong - Response to a ping

Query - Search request

QueryHit - Successful reply to a search request

Push - Request for download

The Ping message has no payload and is used to probe neighbor peers in the network. The message Pong is used in response to a Ping and has information of the peer and some basic information regarding the files this peer is sharing. The message Query is used to search for files, this contains the search criteria and several flags that define limits to how the message is propagated. QueryHit message is the response to a Query that is only used if files corresponding to the search criteria were found. The message contains a list with all the file information that corresponds to the search criteria. Finally there's the Push message, this contains information about the file being requested to that peer so that a download can start. The message also contains the IP and port of the requester to where the file should be pushed. To prevent routing loops all the messages have IDs and will be discarded if it reaches the same peer twice.

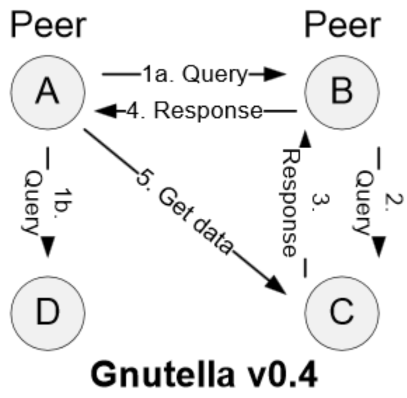


Figure 2.10: Searching and Requesting a File in Gnutella v0.4.

The search in the Gnutella network is done by flooding, also known as constrained broadcast, meaning that the query is sent to all of the peer's neighbors recursively until the TTL reaches 0. All messages in the network have a max TTL of 7, after each message passes through 7 peers it is no longer retransmitted. Flooding is practical for small networks but as every request causes the peer to retransmit the request to every peer it knows, this causes larger networks to exponentially generate more traffic making the solution inherently unscalable.

The evolution of the protocol in Gnutella v0.6 solved v0.4's limitations by going towards a superpeer-based architecture similar to what KaZaA/FastTrack had implemented. After version 0.6 the peers are superpeers or regular peers. This added more structure to the

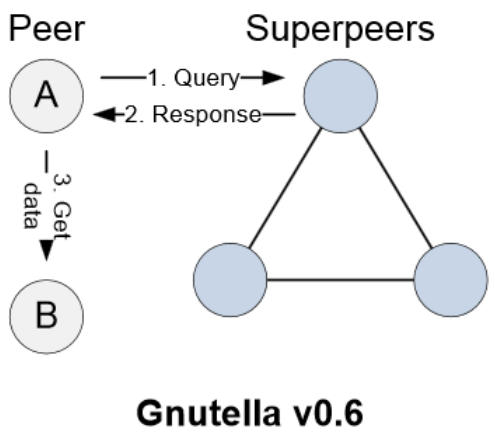


Figure 2.11: Searching and Requesting a File in Gnutella v0.6

network as superpeers are connected to each other and queries are only forwarded to regular peers if they can handle them. Regular peers may turn into superpeers dynamically depending on bandwidth, uptime and CPU power. This way peers are regarded due to capability and regular peers connect to one superpeer using it as a proxy to the entire network. This version of Gnutella also implemented DHT functionality among superpeers for searching content and caching for pong messages, which greatly reduce the signaling traffic and overhead of the network as a whole.

As seen in 2.11, the search became more simple and efficient by using superpeers. As regular peers use the superpeers as proxies to the network, they use less resources. Meanwhile superpeers are able to optimize the network by caching information and relying on a smaller network between superpeers, making the whole architecture more stable and scalable.

Chapter 3

Proposed Solution

This chapter describes the proposed solution following what was explored in the previous State of the Art chapter. With all the objectives in mind, the proposed solution is a pure javascript implementation of a prototype capable of communicating with a BitTorrent network and a WebRTC Network.

3.1 Specification

Several parts of the solution are specified in this section, as well as some of their requirements. As the objectives for this dissertation have been laid out, some specifics of the protocols used forced the solution to be tailored around them, affecting the design choices of the architecture. Therefore it makes sense to define the specifications first, before laying the architecture that was highly influenced by the former.

After this point on the document, the prototype solution of this dissertation will be regarded as *SeedSeer*. The name comes from Final Fantasy XIV, SeedSeers are the authority between elementals and humans, a bridge between both races that are regarded with great wisdom, alike this dissertation's solution that tries to be a bridge to the BitTorrent network and the WebRTC Network. As the two words that form it, *Seed* is related to the BitTorrent protocol meaning the uploader and *Seer* meaning to have vision of the future or foresight.

BitTorrent Network

To find content the BitTorrent clients can use several protocols, Local Peer Discovery (extended by BEP 26[1]), Peer Exchange (PEX Protocol), Distributed Hash Tables (DHT) and Trackers. Trackers were the default way BT clients use to find peers for a given content (represented by a hash), the other 3 protocols were introduced as BitTorrent protocol evolved and grown in usage. Trackers were initially over HTTP, but as usage of BitTorrent grown it became too costly to maintain and scale, so a UDP implementation is today the de facto

[1]http://bittorrent.org/beps/bep_0026.html

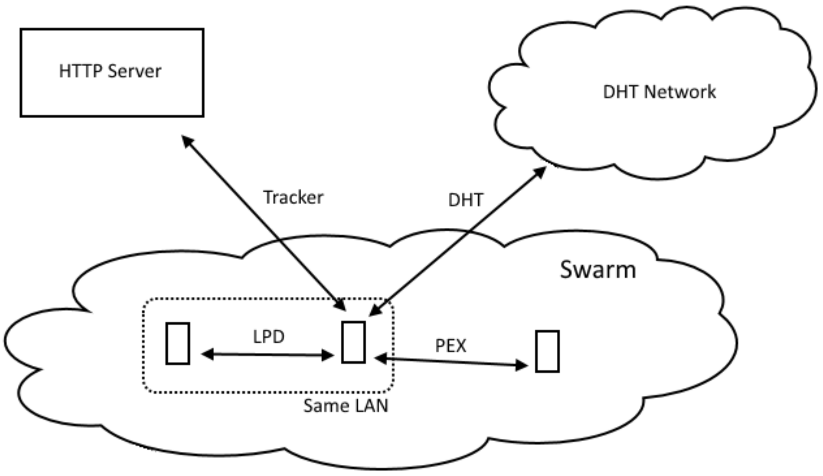


Figure 3.1: BitTorrent Discovery Protocols.

standard of today's implementation of BitTorrent Trackers. With the growth in usage of BitTorrent the use of DHTs became more popular, despite being slower than relying on trackers. Using DHTs provided a decentralized solution that was initially used as a backup way to find peers when trackers were unavailable, but it had become immensely favored since then, so much that some clients today rely exclusively on DHTs to find new peers.

A connection to the BitTorrent network requires the use of either TCP or UDP. The most interesting protocol to find peers for this solution is the DHT, given it is decentralized and scalable, without having to rely on a tracker. While PEX and LPD are interesting protocols to complement the peer list, both aren't reliable for finding the first initial peers. Comparing the use of a tracker to the DHT, the DHT scales better and is resilient to attacks while having the trade-off of being slower than the centralized solution.

Asside from joining the swarm, BT clients use TCP and UDP for downloading and uploading content. This adds the requirement for SeedSeer to be able to use these protocols.

All BitTorrent clients use a configuration on how many connections it can have globally, this limit is used only for actual active peer connections for download and upload purposes, everything else goes out of this limit. Typically the default is around 200 connections, while UDP requests specially for trackers or DHTs don't count towards this limit. As SeedSeer is a client in javascript, the number of available connections might be a limiting factor compared to a native bittorrent client.

WebRTC Network

For a WebRTC connection to be established between two peers, a signaling phase is needed which needs to be handled by a third party. The WebRTC standard let us consider the choice of method or protocol to use for the signaling process. Adding to this, for a peer to join the network it has to know at least one peer. This is a typical problem of a decentralized solution, the client needs to have a defined way to bootstrap a client into the network. For truly decentralized solutions (like Gnutella), the client can scan the internet in an effort to find other peers, this approach is quite slow, relies on clients to run for long periods of time and makes it impossible to bootstrap on clients behind a NAT. On the other hand using a centralized solution goes against the objectives of this dissertation, so the approach should be a middle ground. In summary, the WebRTC signaling has to be reliable and as close to serverless as possible and the bootstrap process (of finding the first peer) has to be a simple process, both for the user and from a tecnical standpoint.

The chosen solution for the signaling process is the WebSockets Protocol. WebSockets are easy to use and are cheap on resources, to use them a HTTP server is required. As the WebRTC signaling is done between two peers through WebSockets, the HTTP server works as a third party, playing as a proxy between the two. As WebSockets are used for signaling, it also becomes the ideal solution for peer discovery, before any WebRTC connection is established.

WebRTC connections are resource expensive to start, they take relatively a long time to establish and aren't efficient to exchange a low number of packets. For this reason WebRTC is used for the transfers, leaving WebSockets essentially for peer discovery and the signaling process. By transfers, it is meant that WebRTC connections handle the data, the metadata and individual requests for parts of the files (pieces), as the client already has an established connection to the required peers.

Arguably WebRTC could be used for peer discovery, using WebSockets for the initial signaling and then handling everything over WebRTC. There are two reasons not to do this. The first is that clients become lighter, clients don't need to be so complex or have too many connections, most clients will be connected to the network for short periods of time and using only a browser. So relying on a chain of peers that are totally decentralized would fail to give a good service to most of the peers. This problem existed in earlier versions of Gnutella, with the added complexity that when the chain of peers were broken, establishing new WebRTC peers to de-fragment the network would be increasingly more difficult as the clients can't simply ping or try to connect to new peers over WebRTC like Gnutella would do, by simply using UDP. The second reason relies on limitations on today's solutions this dissertation is built upon, currently the browsers have limits on how many WebRTC connections are allowed, these are set because WebRTC connections are heavy to establish and have a lot of overhead if we consider it for low amount of packets per peer.

As referred in 3.1, BitTorrent's connection limit per client is typically 200, however UDP communication doesn't count towards this limit. Similarly, browsers have a limit of WebRTC connections. As WebRTC will only be used for transfers it is similar to the active TCP connections in BitTorrent clients, furthermore WebSockets are used for everything else, just like BitTorrent uses UDP.

Heterogeneous Peers

The advantage of a pure javascript solution is the ease of use for new users. As no installs are required, the client is served directly on a browser. This lowers the learning curve that nowadays stands as a barrier for non-techsavvy users to use p2p solutions like BitTorrent clients. However a javascript client served through a webpage has several limitations. This would not provide most of the functionality required for SeedSeer, so to appropriately achieve all objectives, a heterogeneous architecture was chosen.

Similarly to Gnutella, SeedSeer has two types of peers, the Supernodes and regular Nodes (regarded as simply "peers" from now on). While in Gnutella peers can become supernodes depending on their resources, in this solution it depends on where the client is run. Peers run the client as a regular webpage, while supernodes run the client essentially like a browser extension , exploring other browser resources that are unavailable to simple JS running in a website.

There's mainly 3 reasons in this solution for using a Supernode:

- Bootstrap

  Initially a peer has to join the network, this is commonly referred as bootstrapping. While bootstrapping is essential, not all peers require the ability to bootstrap other peers, as most clients would be ideally lightweight, have low performance and be behind a NAT which makes them unreachable. In SeedSeer the peers will bootstrap through a supernode.

- UDP/TCP Capabilities

  The capability to use UDP and TCP is essential for several functionalities, like connecting to the BitTorrent network. However regular peers running javascript can't use these protocols directly. Using supernodes for this permits them to handle the BitTorrent network, while bringing its content to other peer that can't access it.

- Router for WebRCT Signaling

  As referred in 3.1, a third-party is required to establish a WebRTC connection. To avoid using an external entity, a supernode handles the routing between two peers that want to connect using WebRTC.

Identically to most supernode architectures, each peer is only connected to one supernode. An example is shown on fig. 3.2, the Supernode is shown as the hexagram and regular peers as circles. All peers will always keep an active websocket connection to a supernode (shown as normal lines). Both supernodes and peers run the exact same javascript modules, being that the supernode has more modules to handle all the extra functionalities. This avoids having to maintain different versions of code for both types of peers.

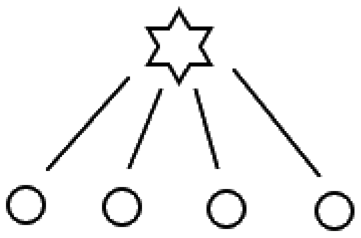


Figure 3.2: Four peers connected to a Supernode.

Chrome App

For a supernode to run in a browser there are two possibilities, it can run as an *extension* or as an *app*. An extension typically extends the browser behavior and modifies web pages. An app however runs with a dedicated user interface (UI) and can run in the background, without the requirement of having a certain web page open.

The supernode implementation runs as a Chrome App, with the added benefit of having access to the Chrome API in full. The Chrome API gives access to storage and network capabilities, with this the supernode can manage where the content is stored and use the network API to have access to sockets.

The peer's code that includes the visual interface runs also within the supernode. However as a chrome app, it has slight differences due to its UI not being exactly a webpage.

Identity-based Routing

As peers rely on supernodes for establishing WebRTC connections, identifying peers by their IP address wouldn't have any benefit, as they can't connect to them directly. While supernodes know exactly the peers and their routing info, all communication to peers is always focused on their identity (PeerID) or partial identity (nick). As there is no type of authentication, registration of identities or even a way to assure that identities are unique between different SNs, the same identity in a different time might refer to a different peer. Identities are randomized and don't have a high chance for collisions to be common, if a collision happens within a supernode, the supernode is able to solve it.

All communication in Seedseer is possible using these identities, while actual p2p connections through WebRTC does expose the clients information, all the other interactions between peers and even the indexed content uses identities for routing the messages. This abstraction provides some privacy, as only supernodes can translate identities into the actual clients information (like IP addresses).

3.2 Architecture

This section defines the proposed architecture for Seedseer in accordance, but not limited, to the specification in the Section 3.1 above. The architecture section is divided in three main parts, the *Seedseer client* that specifies the architecture of a regular peer, the *Seedseer Supernode* that specifies the architecture of a supernode that also includes the previous client, and finally an *overview of the network* and how several peers communicate between one another. Lastly, some security concerns are pointed out about the architecture given the P2P nature of the solution.

SeedSeer Client

The Seedseer client runs in a browser as a webpage and constitutes a peer in the Seedseer network. The client in pure javascript is capable of using Websockets and WebRTC to transfer data and consume content directly in the browser. A regular peer is limited to these two protocols, so it will only stay connected to the WebRTC Network referred in 3.1.

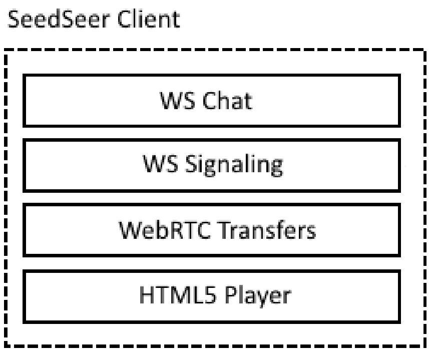


Figure 3.3: SeedSeer Client Architecture.

Fig. 3.3 depicts the Seedseer client and the following four modules:

- WS Chat, responsible for handling communication between the users. Used for communication between active peers, this module is used for it's chat functionality, command input and notifying all peers about changes in the network (e.g. new content).
- WS Signaling, this module does the WebRTC Signaling required to establish a WebRTC connection. This is done through Websockets, using the connected supernode as a proxy between the local peer and the desired peer.
- WebRTC Transfers, manages all file transfers between peers with an active WebRTC connection. The module can request new connections or receive requests initiated by other peers. Connections are maintained as long as one of the connected peers is downloading available content from the other.

- HTML5 Player, plays content directly in the client. As long as the content is supported, the player can be used to play local content that being shared or content downloaded from other peers.

The client also handles the interface that triggers all the functionalities. Some functionalities can be disabled or used exclusively by a supernode, thus the interface is versatile as long as it is linked to a known functionality. Although all functionalities can be used from the interface, command input is also supported.

SeedSeer Supernode

Supernodes (SNs) are clients with extra capabilities when compared to regular peers. Despite running pure javascript like the regular peers, the code of a supernode runs inside a Chrome App, as referred in 3.1. Using the Chrome API, a SN has access to sockets which are able to communicate through TCP and UDP, which unlocks the possibility to create a HTTP and WS Server and also to use the BitTorrent protocol. Peers always connect to only one SN through websockets, which is their gateway to the whole overlay network (as referred in 3.1).

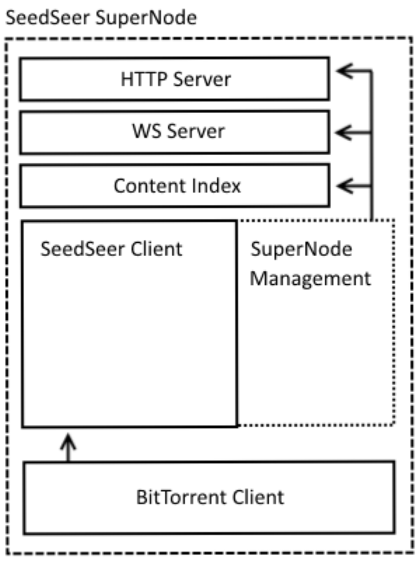


Figure 3.4: SeedSeer Supernode Architecture.

Seedseer Supernode is composed by the Seedseer client 3.2 and also for 5 other modules:

- HTTP Server, allows the peer client to be served through HTTP. This serves webpage with javascript code that is required for a regular peer, with it goes the configuration of the supernode, making the client connect to the SN through Websockets after it loads.

- WS Server, websocket server that manages connected peers. This module is used for all functionalities that require Websockets, identity routing is used outside of this module, whereas the WS Server encapsulates the peers information.

- Content Index, this module that manages the known content of the network and which peers are part of each swarm.
- SuperNode Management, responsible for aggregating all module functionalities and how they are accessed. This is the main module of the supernode, it is responsible for handling all SN functionalities and the BitTorrent client.
- BitTorrent Client, accesses the BitTorrent network downloading new content into the WebRTC network. The module adds it's interface to the DOM and is managed by the SuperNode Management module.

Overview of the Network

A supernode is essential for regular peers to exchange content and to get access to content in the BitTorrent network. Fig. 3.5 shows an overview of the different networks from the point of view of a supernode. A supernode has peers connected to it through websockets, these peers use WS to establish WebRTC connections and share content. As the supernode is also a peer, it can also join a swarm and share content with other peers. The supernode can also connect to other supernodes through WS, essentially for searching for content and for establishing a proxy between peers connected to different supernodes. Lastly, the supernode has a BitTorrent client, so it connects to the BitTorrent network to share content, being that the main motivation is to bring content to it's WebRTC network.

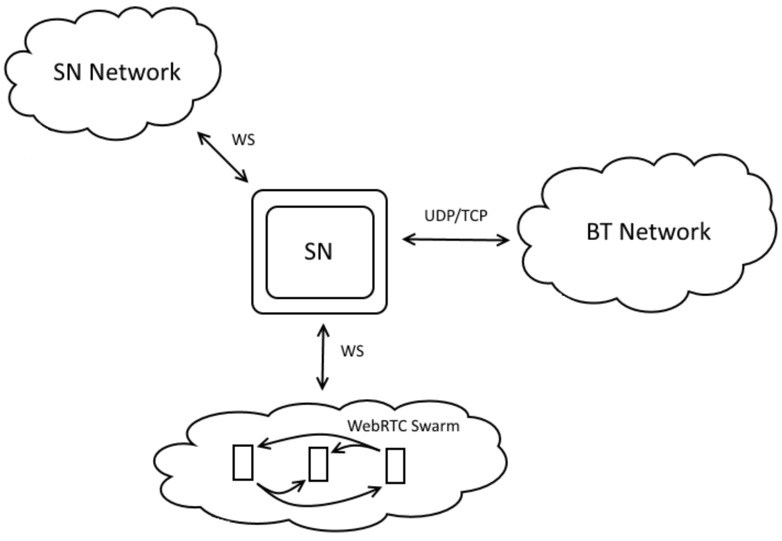


Figure 3.5: Overview from the point of view of a SuperNode.

The supernode is used to get the list of peers in a swarm and also for establishing the connection between peers. Peers can be in several swarms sharing different content, this is all handled through WebRTC and the supernode is not involved after a connection is established. When trying to join a swarm, a request is sent to the supernode that replies with the list of peers in the swarm. After doing so, the client can request to establish connections to those peers, this WebRTC Signaling is also done through WS like referred in 3.1.

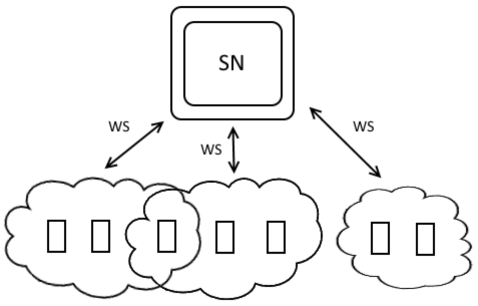


Figure 3.6: Different swarms under a SuperNode.

As it is depicted in fig. 3.6, peers can be in more than one swarm at once. If a peer joins a swarm and has already a WebRTC connection to some of the peers, there is no need to do WebRTC Signaling using the supernode, the peers just create a new datachannel without the need to establish a new connection. The supernode is also a peer, thus it can be part of the swarms, the only noticeable difference is that it does the WebRTC Signaling with itself and the other peer.

Chapter 4

Implementation

The protocols for the integration of this architecture were introduced in the previous chapters. This chapter describes the implementation of such an architecture, individually describing what are the roles of each module and how they are constituted. The aim of this solution is to connect via BitTorrent and WebRTC, sharing multimedia content stricktly through a browser. To keep the clients light, only pure javascript is used, with no external libraries, being the only exception the implementation of the BitTorrent client described in 4.2 that is present in supernodes.

4.1 Bootstrap

There is no automatic way to find supernodes, the user is responsible for finding a supernode and connect to it so the peer can bootstrap. While seedseer was being developed, a public list with supernodes would advertised, these supernodes had DNS addresses or a static IP. For this public list, a script would scan known IPs of machines that had supernodes running. This was done using the the “detect.js” from that IP, if it was successful it would mean the supernode was online and free to be used. This method also allows the a SN to report its current load or refuse to respond if the SN is already at its max threshold of connected peers.

For peers to be able to bootstrap, the supernodes should never be behind a NAT and be accessable directly from the Internet. As the peers only require a connection to a supernode to bootstrap, the process is finished as a websocket connection becomes stable with a supernode. Unlike other P2P solutions, the peer only requires this connection, further connections to other peers are only established when the peer wants to share content and effectivelly joins a swarm.

4.2 Components

As was discussed in chapter 3, the supernode is defined by several components. In fig. 4.1, three components are colored as grey. These components use external code from github projects, where they were then integrated into the whole solution that constitutes seedseer. The components HTTP Server and Websocket Server are described in 4.2 and the component BitTorrent Client is described in 4.2.

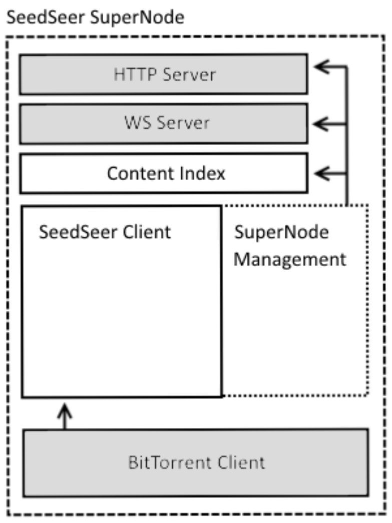


Figure 4.1: Supernode Architecture.

WebSocket Server

This component is pure javascript HTTP and WebSocket server created by Google[1]. Similar to NodeJS implementations for the same service, this project provides a lightweight server that runs as a Chrome App in Chrome, taking fully advantage of the Chrome API. To provide this service it uses the Chrome API to have access to a TCP Server and have access to sockets. Most of the code is kept in the HTTP.js file, while the rest are mostly basic files required for chrome apps. As WS is a full-duplex protocol over TCP that is stripped out of HTTP, it comes with the HTTP functionalities that could be disabled for this dissertation, but were kept for serving client code for other users and for detecting supernodes. This project also came with an implementation of SHA1 (in SHA1.js), which was useful for hashing content for the WebRTC network, removing the need to take another external library implementation to do the hashing.

The only modification done to this component was a fix for retrieving the User Agent of browsers. As a split was being done on the ":" character, this meant that some version of firefox were getting cut incorrectly. For example, "Mozilla/5.0 (Windows NT 6.1; WOW64;

[1]https://github.com/GoogleChrome/chrome-app-samples/tree/master/samples/websocket-server

rv:31.0) Gecko/20100101 Firefox/31.0" would incorrectly be read as "Mozilla/5.0 (Windows NT 6.1; WOW64; rv".

A simple fix was applied and the issue reported on the official repository. [2]

BitTorrent Client

This component comes from an open-source Chrome App named JSTorrent[3], that stands for "JavaScript Torrent". JSTorrent is a pure javascript solution that implements a BitTorrent client over Chrome, using dependencies like Underscore, jQuery and web-server-chrome[4].

This component is used in supernodes to access the BitTorrent network and download content from it. The project is not modified, but the Supernode Management modifies some of it's objects in runtime so it is able to run inside of the Seedseer Chrome App.

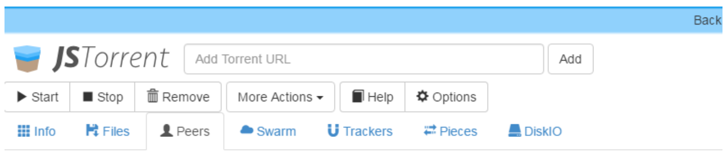


Figure 4.2: JSTorrent running in Seedseer.

Fig. 4.2 shows JSTorrent running inside Seedseer, where buttons were redifined and the "Quit" is changed to "Back", that in this case returns to the regular supernode interface while keeping the BitTorrent client running in the background.

[2]https://github.com/GoogleChrome/chrome-app-samples/issues/237
[3]https://github.com/kzahel/jstorrent
[4]https://github.com/kzahel/web-server-chrome

Code Structure

SeedSeer.js The main file responsible for most of the peer's code.

Util.js Auxiliary functions for the peers.

SN.js The main file for the SuperNodes.

SNutil.js Auxiliary functions for the SNs. sha1.js Library for SHA1 hashing.

http.js HTTP Server over JS, WebSocket capabilities included.

launch-app.js Launcher for the BitTorrent extension.

detect.js Simple script for SuperNode detection.

Listing 4: Code Structure

For peers the code is runs minified, security-wise minified code isn't more secure but it does make it a bit harder to understand and being able to modify javascript code, so it was used as principle from the start. As peer's code was supposed to be as small as possible, minifying the code also made sense especially in this regard.

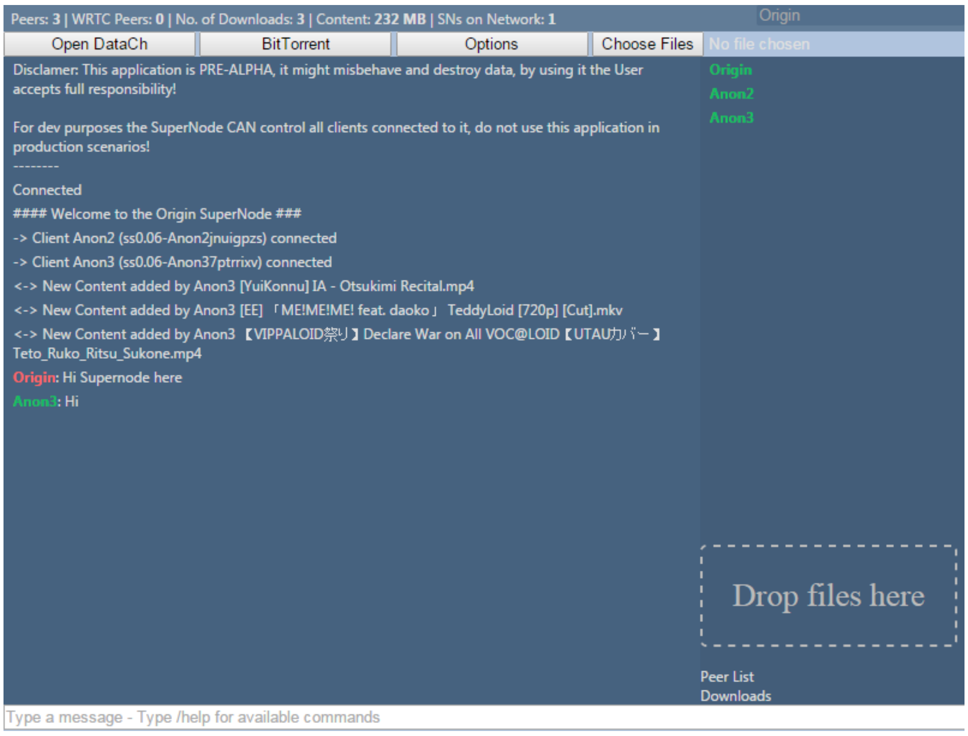
To minify javascript code two tools were tried, JSMIN and YUI, after some tests YUI consistently pulled smaller file sizes so it was chosen as the default minifying tool. Two scripts can be found on the prototype code that allow all the code to be minified by this tools respectively.

4.3 User Interface

Inspired by the previous IRC implementation and by Volafile, the structure of the Seedseer is similar to what one would find on an IRC Application. There are no channels, every supernode maintains one chat room where every peer can chat. As the supernodes are running on a chrome browsers and their capabilities would be limited to the hundreds of users per supernode, it is envisioned the supernodes to be as communities. As a service with a limited number of users per supernode, it is compelling to keep thematic content in a community, for peers directly connected to that supernode. This was the inspiration that influenced the user interface conseptualized for Seedseer.

Figure 4.3 shows the default interface of a supernode. On the right side there's the peerlist that has all the peers currently connected to the supernode. This right side can be switched by clicking on different options on the bottom, this list can be changed depending on the settings in the supernode, removing default options like the Peer list or the Downloads, or adding new functionalities.

On the top is the status bar, "Peers" are the number of total peers connected to the supernode, "WRTC Peers" is a local variable, it states how many WebRTC connections exist



```
Peers: 3 | WRTC Peers: 0 | No. of Downloads: 3 | Content: 232 MB | SNs on Network: 1
Disclamer: This application is PRE-ALPHA, it might misbehave and destroy data, by using it the User
accepts full responsibility!

For dev purposes the SuperNode CAN control all clients connected to it, do not use this application in
production scenarios!
--------
Connected
#### Welcome to the Origin SuperNode ###
-> Client Anon2 (ss0.06-Anon2jnuigpzs) connected
-> Client Anon3 (ss0.06-Anon37ptrrixv) connected
<-> New Content added by Anon3 [YuiKonnu] IA - Otsukimi Recital.mp4
<-> New Content added by Anon3 [EE] 「ME!ME!ME! feat. daoko」 TeddyLoid [720p] [Cut].mkv
<-> New Content added by Anon3 【VIPPALOID祭り】Declare War on All VOC@LOID【UTAUカバー】
Teto_Ruko_Ritsu_Sukone.mp4
Origin: Hi Supernode here
Anon3: Hi
```

Figure 4.3: Supernode User Interface.

open in this client, "No. of Downloads" is the number of unique hashes of content exists in the network, "Content" is the reported size of all the content in the network and "SNs on Network" is how many supernodes are connected to this supernode.

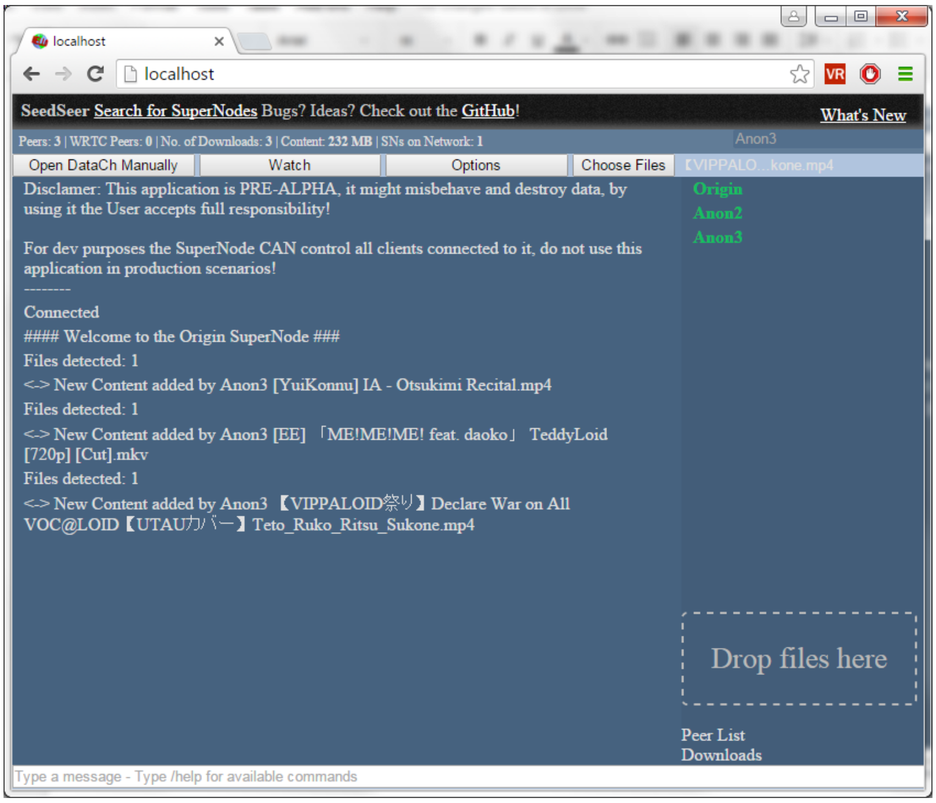
Below the status bar we have the buttons bar, "Open DataCh" forces a WebRTC connection to the peer assigned, "BitTorrent" pops the BitTorrent functionalities, "Options" pops a new window that shows the change log, a feature list, settings and a way to report bugs. Lastly, "Choose Files" allows adding new content to the network.

On the top right corner there's a textbox with the current peer nick, that is bound to a peer identity, writing on this textbox and pressing *enter* changes the nick and peerID.

The peer UI shown in fig. 4.4 is pretty much identical to the UI of a supernode, with two major differences, the top black bar and the "Watch" button instead of the "BitTorrent" button. The black bar has 3 links, the first one is a link to a site that scans for supernodes[5], the second is a link to GitHub[6] where users can report bugs or issues and the last one shows the change log, a small list of features that were changed in the two to three versions.

The watch button pops a black semi-transparent layer that has the HTML5 player, shown in fig. 4.5. This player allows the peer to watch videos or play music. The type of content that can be played this way is hard to detect in javascript, as there is no assured way to check if the computer that is running the peer has the required codecs or not. The ability for the HTML5 player to work properly depends in the browser and installed codecs being used.

[5]http://seedseer.pt.vu/
[6]https://github.com/SeedSeer/SeedSeer



```
Peers: 3 | WRTC Peers: 0 | No. of Downloads: 3 | Content: 232 MB | SNs on Network: 1
```

```
Disclamer: This application is PRE-ALPHA, it might misbehave and destroy data, by
using it the User accepts full responsibility!

For dev purposes the SuperNode CAN control all clients connected to it, do not use this
application in production scenarios!
--------
Connected
#### Welcome to the Origin SuperNode ###
Files detected: 1
<-> New Content added by Anon3 [YuiKonnu] IA - Otsukimi Recital.mp4
Files detected: 1
<-> New Content added by Anon3 [EE] 「ME!ME!ME! feat. daoko」 TeddyLoid
[720p] [Cut].mkv
Files detected: 1
<-> New Content added by Anon3 【VIPPALOID祭り】Declare War on All
VOC@LOID【UTAUカバー】Teto_Ruko_Ritsu_Sukone.mp4
```

Figure 4.4: Peer User Interface.

The most noticeable difference is on videos that use the H264 codec, where Firefox doesn't support it for licencing reasons, while Chrome does.

The way to select content to be played was kept simple. The last content which was interacted with, is available on the player, whether it is able to be played or not. As the HTML5 player is a component, it's layer can be hidden. This allows the user to start playing a video or music, hide the player but keep hearing the audio in the background. Despite being really simple, user feedback mentioned it as a positive aspect, specially by those that would use it as a music player while chatting.

Every peer can add content to the network, the supernode will keep the filename, hash and size of the content and add the original peer to the swarm list. When a peer connects to a supernode, it gets the full list of content that is available on the network, after that any further content that is added will be advertised on the chat room like it is seen on fig. 4.4. To see content that is available to download, a left click on "Downloads" in the right bottom corner will switch the right side of the UI for all the filenames of the content that is on the network, like it is shown on fig 4.6. By clicking on any of the content in this list, the peer requests the peer list from the supernode, joins the swarm and begins downloading the chosen content from the available peers.

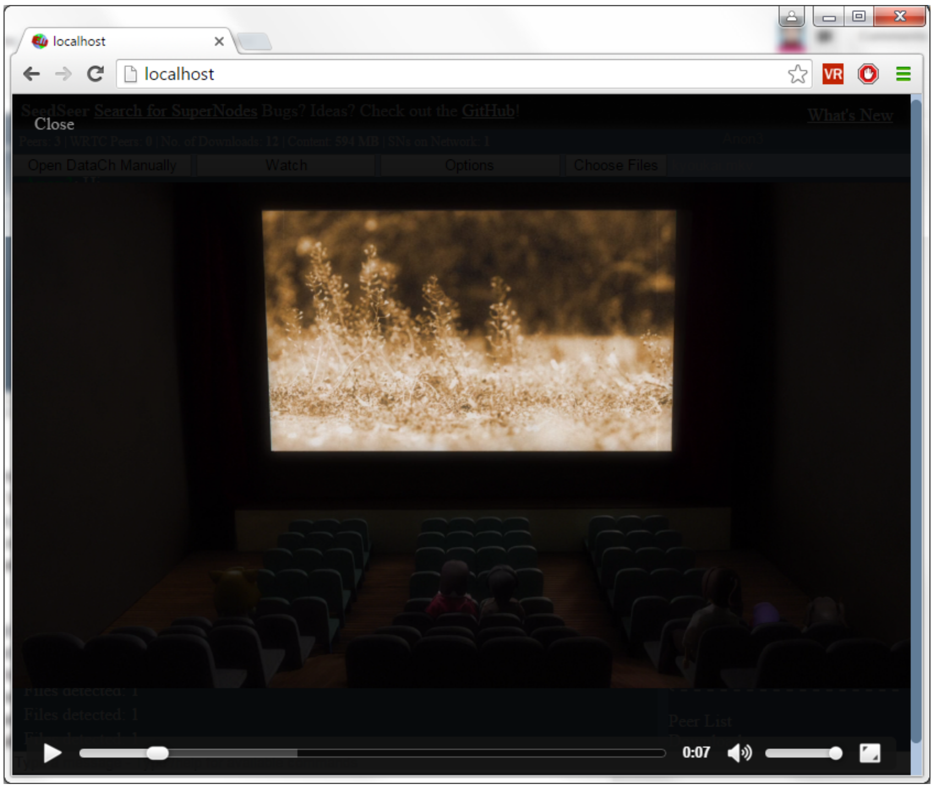


Figure 4.5: HTML5 Video on a peer.

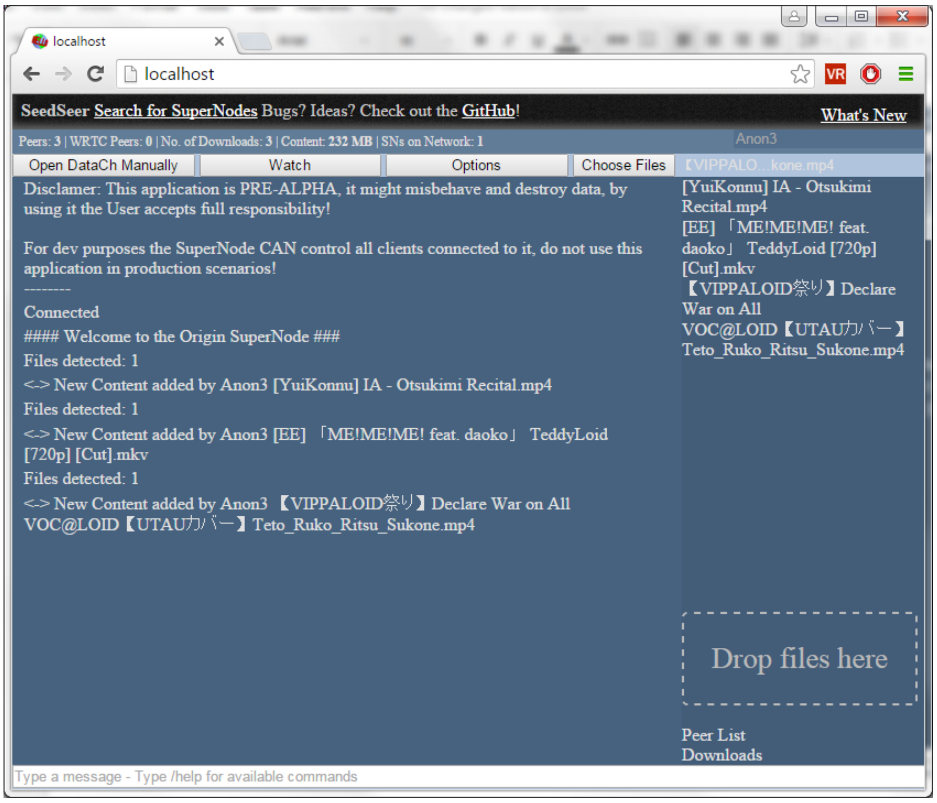
4.4 WebSockets Functionalities

Packet validation

Every packet that reaches a supernode is verified to check if it is well-formed. On certain types of packets extra fields are ignored allowing extra business logic on the peers that supernodes might not know of. Some limits should be specified to avoid possible DoS attacks on supernodes and on peers.

Adding content & Swarms

For content to be added to the network it has to be hashed previously. To do this the peer loads the file into memory and hashes it using SHA1's library, the result is then sent to the supernode along with filename and file size, if the content is new the supernode saves all info and creates the swarm with the peer as the only seeder, if the content already exists all info is ignored and it adds the peer into the swarm.



SeedSeer Search for SuperNodes Bugs? Ideas? Check out the GitHub!

Peers: 3 | WRTC Peers: 0 | No. of Downloads: 3 | Content: 232 MB | SNs on Network: 1

```
Disclamer: This application is PRE-ALPHA, it might misbehave and destroy data, by
using it the User accepts full responsibility!

For dev purposes the SuperNode CAN control all clients connected to it, do not use this
application in production scenarios!
--------
Connected
#### Welcome to the Origin SuperNode ###
Files detected: 1
<-> New Content added by Anon3 [YuiKonnu] IA - Otsukimi Recital.mp4
Files detected: 1
<-> New Content added by Anon3 [EE] 「ME!ME!ME! feat. daoko」 TeddyLoid
[720p] [Cut].mkv
Files detected: 1
<-> New Content added by Anon3 【VIPPALOID祭り】Declare War on All
VOC@LOID【UTAUカバー】Teto_Ruko_Ritsu_Sukone.mp4
```

```
[YuiKonnu] IA - Otsukimi
Recital.mp4
[EE] 「ME!ME!ME! feat.
daoko」 TeddyLoid [720p]
[Cut].mkv
【VIPPALOID祭り】Declare
War on All
VOC@LOID【UTAUカバー】
Teto_Ruko_Ritsu_Sukone.mp4
```

Figure 4.6: Showing content available to download on a peer.

4.5 WebRTC Functionalities

For an easy setup of multiple WebRTC connections a group of functions were crafted so it would make the process more effortless. First there's the functions that handle WebRTC connections, those are NewWRTC, StartWRTC, MatchWRTC and CloseWRTC, then there is an object that handles all signaling components that are over WebSockets, that is the "SignalingChannel" object, and finally there are the functions that handle the data channels themselves, those are createStatusWRTC, bindStatusWRTC and createDchanWRTC. All of these will be extensively explained in the next sub-chapters.

Connections

For WebRTC to work the first step is on NewWRTC function, it creates a RTCPeerConnection Object with a given configuration. There are many ways to configure a RTCPeerConnection object, but as this dissertation will only explore data channels, the arguments passed to the constructor are usually the same. First is the STUN servers, STUN means "Session Traversal Utilities for NAT" and as the name states it allows for the client to know more about itself, as most computers today stand behind NATs, the client needs the STUN server to know it's own public IP so it can advertise itself so other external clients can find it even behind a NAT. There are several STUN servers available to the public. Depending on the browser being used the STUN servers used differ, using google STUN servers if it's on chrome or another one otherwise. The other parameters are standard, "DtlsSrtpKeyAgreement" is set to "true" as it is the only way WebRTC works across-browsers and "RtpDataChannels" to "true" as it is needed for data channels.

To avoid multiple connections to the same peer, the peer identity is set on the object and the WebRTC object is assigned to a global array of WebRTC connections. After that a *SignalingChannel* object is created, this object is responsible to handle all the signaling process over WS until the connection is open and ready to use, and the required data channels are created. A callback is also assigned to handle ICE Candidate packages.

NewWRTC is only used to create all the objects needed to WebRTC. To start the establishing process, the initiating peer uses StartWRTC function. This function handles the creation of a SDP Offer while the peer that receives this uses the NewWRTC to create all necessary objects and uses MatchWRTC to create the SDP Answer. After these are exchanged both clients send and gather ICE Candidates until they are able to open the connection. ICE Candidates mean Interactive Connectivity Establishment, it's a hacky way of reporting nodes on the network so that the other peer that is outside the network is able to use that route to communicate. The same is done for both peers at the same time until the browser decides that they received enough ICE Candidates to know how to communicate effectively with the other peer. There is no need for all ICE Candidates to be able to establish the connection. Depending on the browser being used, many strategies have been used to exchange ice candidates, one of the popular ones was to send only one packet with the whole ICE Candidates that the peer was able to gather, however after many tests it was confirmed that as not every ICE Candidate is needed, sending multiple ones as they are getting generated would reduce the time needed to open the WebRTC connection.

In chrome, after the connection is open, the ICE Candidate callback is set to auto-destroy itself, as further ICE Candidates aren't needed and in some cases strangely formed ICE Candidates would provoke odd behaviors, especially between different browsers or versions.

As a default, a data channel is created named "#status" this data channel used as a basis for data transfers.

Transfers

To make transfers, the default #status data channel is used for direct request between two peers, and for transfers a data channel is created with the content's hash as the channel name, this makes it simple to handle multiple streams of data even if they are from the same peer and yet keep them easily organized. Creating new data channels after the connection is open is just a matter of calling the function "createDchanWRTC" with the desired hash of content, though it is currently requiring new SDP and ICE exchanges, it doesn't seem to affect the connection that is open. The transfers are done in chucks, the content is divided in pieces a bit like BitTorrent and then are sent through its assigned data channel, when the file is complete on the other end it is requested to be sent to the disk.

The transfers are uncomplicated, everything is handled in memory. At some point, chucks are saved to the disk using Chrome API, but this is only suported on a supernode. While in a peer the contents can only be saved to the disk when the download is completed.

The algorithm of requesting chucks to be downloaded is straightforward, it always goes by order, if there are multiple peers if chooses a multiplier of chucks and requests them randomly to any peer that has them. The number of chuck groups, that is similar to *pieces* in BitTorrent, is set when a connection is established. While in BitTorrent this is set in the metadata in the torrent file, in seedseer it isn't a static value.

4.6 Commands

For ease of use, some commands are provided for both peers and supernodes. Listing 5 shows the commands and parameters that are available in SeedSeer.

Peer Commands:
/re(load) Reloads the client and forcibly drops all connections, if the client is being loaded from the supernode and not locally, it loads a new version of the peers code and reconnects to the supernode.
/nick <nick> Changes the nick to the one specified and renews peer identity. All routing info on the last identity is discarded, WebRTC connections are maintained but new connections can only be done over the new identity.
/clear Clears the chat window.
/mod <Module Name> <command> This allows the peer to interact with a given module, this is used when the module functionality happens or is triggered on the supernode. It isn't required if the functionality is local or supported by the module's UI. The command sent is module specific and will be sent to the SN, the result of it depends on the module.

SuperNode-Only commands:

/re(load) (p) Reloads SN and reloads all connected peers, if "p" is added only peers will be requested to reloaded.
/list Lists information on every peer directly connected to the supernode, for this the SN requests all connected peers for their browser and geoIP information to complement identity info that the SN already has. The info is shown like this: <order> : <nick> (<peerID>) <browser> <country>
/sw Lists all swarms and their peer lists.
/module <Module Name> (enable|disable) Supernodes can have default modules enabled, this command allows the supernode to dynamically enable or disable modules. If a new mod is enabled all connected peers will receive the module and can use it after that. More about this on further chapter.

Listing 5: SeedSeer commands and parameters.

Some supernode commands behave similarly to a C&C used in a botnet, the supernode can force peers to do certain actions. This only happens when the peers load it's client from the supernode directly, which is discouraged for security reasons. This functionality of loading the client's code from a supernode was only kept because of it's ease to test multiple clients at the same time and for it's simplicity to load new code into the peers. This functionality should never be used outside of a test environment.

4.7 Supernode Settings

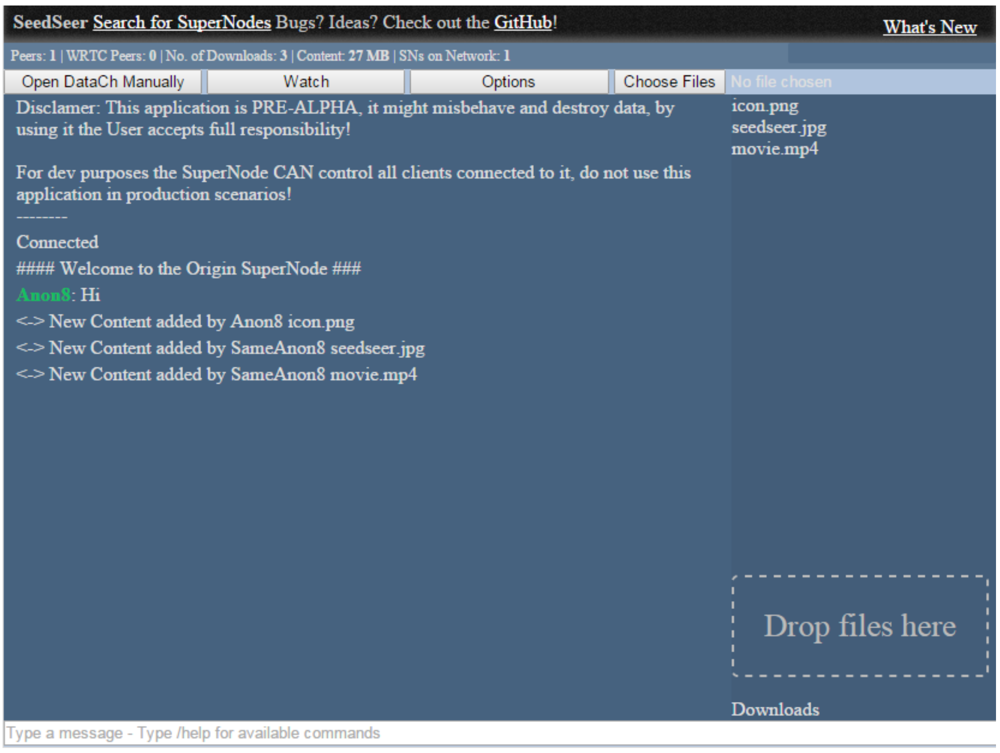
To allow customizability to the supernodes, settings can be changed to appeal to several different communities, supernodes can also change css and styles overall. Asside from theme changes, there are two settings available, *hidden content* and *hidden peers*.

Hidden content is one of the options, this means that peers can't see what content is available and won't be notified about new content, they will however be able to join the swarm if they know the hash for the content. This might be useful for communities that focus on rare content or on content that is by itself controversial.

Hidden peers is another option, the supernode can make the peerlist unavailable, as well as not exposing peer connections, identity changes or connects/disconnects are shown. The total number of peers is also hidden. This is hidden information is on the supernode level, so peers have no way to request this information. Peers are still allowed to use the chat functionality, meaning that their nicks will be visible at that time. However, similarly to anonymous forums, as every peer is free to change identity without anyone knowing, peers remain anonymised to a certain extent.

As the chat room isn't P2P, they rely on the supernode to relay the messages, anonymity is maintained as long as the supernode can be trusted. For file sharing however this isn't the case, as WebRTC exposes routing information.

As an example, fig. 4.7 shows a peer that joined a supernode with hidden peers option enabled. There is no peer list and the peer doesn't know how many peers are currently connected, content however is visible. Only one peer talked on the chat and it seems that different peers added content to the network, but in fact all the content was added by the same peer. As every peer can change identity freely, the amount of peers connected is unknown. In 4.7 there were 8 peers connected, but there is no way for the peers to get that information.


SeedSeer Search for SuperNodes Bugs? Ideas? Check out the GitHub!
What's New
Open DataCh Manually
Watch
Options
Choose Files
No file chosen
Drop files here
Downloads
Type a message - Type /help for available commands

Peers: 1 | WRTC Peers: 0 | No. of Downloads: 3 | Content: 27 MB | SNs on Network: 1

icon.png
seedseer.jpg
movie.mp4

Disclamer: This application is PRE-ALPHA, it might misbehave and destroy data, by using it the User accepts full responsibility!

For dev purposes the SuperNode CAN control all clients connected to it, do not use this application in production scenarios!

Connected
Welcome to the Origin SuperNode
Anon8: Hi
<-> New Content added by Anon8 icon.png
<-> New Content added by SameAnon8 seedseer.jpg
<-> New Content added by SameAnon8 movie.mp4

Figure 4.7: Example with Hidden Peers option enabled.

Chapter 5

Results

5.1 Survey

In this subchapter the data collected from an online survey will be presented. This survey realised in the scope of this dissertation had two goals in mind, to better understand how internet users consume content online and to assert from the data if the concepts applied on seedseer seem relevant to the users.

The response rate on this survey is hard to determine due to lack of analytics from Google Forms, so to gather data on conversion rates, this survey was posted on multiple online communities through an URL Shortener where the following data can be examined:

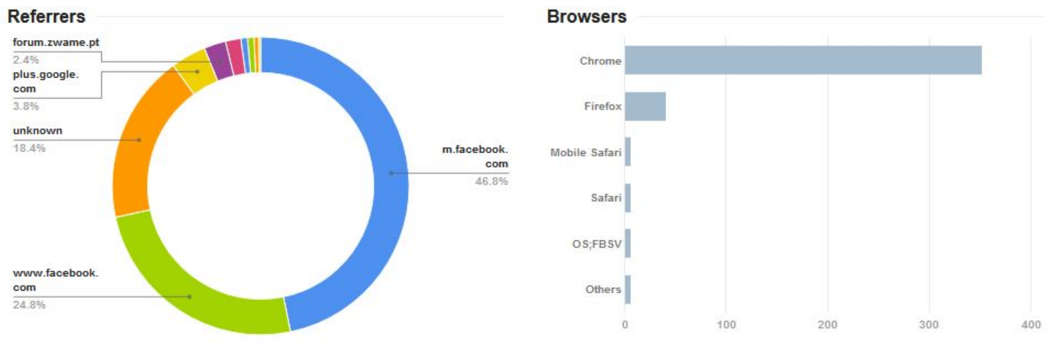


Figure 5.1: Referrers and Browsers used.

To the day of writing this, the survey got 378 clicks through the URL Shortener link[1]. As seen in fig. 5.1, 71,2% of the users came from Facebook, 47.1% of which came from mobile. Other referrers like Google+, Reddit, Zwame and Twitch had very low click rates. Some of the "unknown" referrers are presumably from people to whom the link was shared directly, therefore leaving no referral data. 318 of the users that came through the link used Chrome as a browser, and 37 used Firefox, being that only less than 7% used another browser.

[1]https://goo.gl/#analytics/goo.gl/forms/RerUjdZyidSWJIiN2/all_time

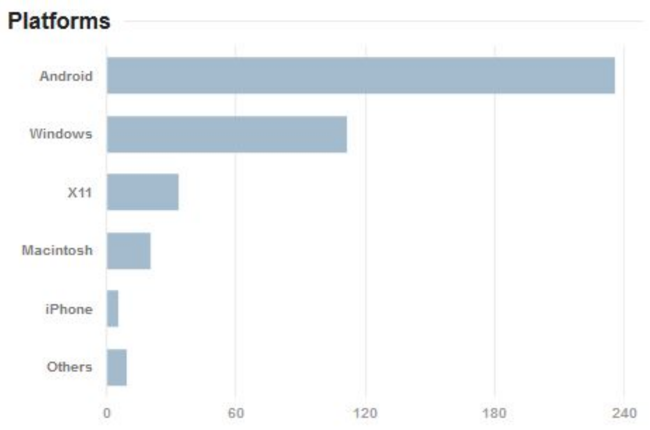


Figure 5.2: Platform used.

As reported above, most of these users came from mobile, 57% of all clicks through the URL Shortener came from either Android or iOS, which isn't surprising as 47% of the total clicks came from the Facebook Mobile domain alone.

Based solely on the URL Shortener the conversion rate, meaning the rate between a user completing the survey in contrast to only clicking on the survey, would be of 31%. However the URL Shortener wasn't used on most of the promotion in Google+, due to shortened links being shown as unappealing and suspicious in that platform. It is unclear if the conversion rate was actually lower or higher, but it is certain that some responses did come from Google+, most of those from users outside of Portugal.

The survey was advertised in several sites and communities, both in Facebook and Google+, the focus was to share the survey in communities ("groups" in Facebook) related to Streaming, P2P or content consumption in general. In Facebook the survey was also shared in the group of University of Aveiro and related, which is why most of the replies to this survey are from Portugal. The same these approach was used for Reddit and YCombinator, however as it was posted with a URL Shortener under a new user and both use upvote systems, the number of clicks were almost zero, the same was repeated without the URL Shortener, but based on the time of the posts and the survey reply timestamps it didn't seem like many results were obtained, if any. Lastly, the survey was also shared in an online game (FFXIV) and during Twitch streams, while the number of clicks were low, these had very high conversion rates.

Due to the choices stated above, the data may be a bit biased towards tech savvy users and university students, as these two are predominant in the communities where the survey was shared.

The survey was answered by 118 people, 94 male (79,7%) and 24 female (20,3%). To have a better understanding of demographic of the respondents fig. 5.3 shows the age and gender of all of them. Most of the respondents are between 18 and 25 years old on both genders, 50

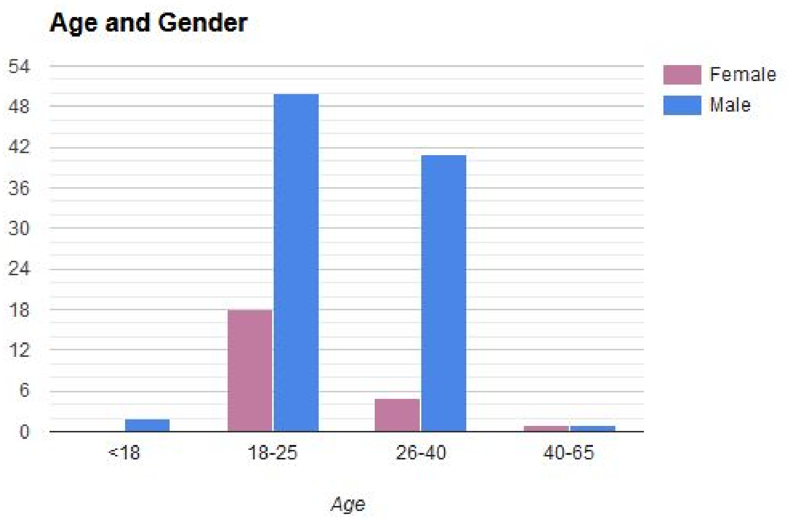


Figure 5.3: Age and Gender of the respondents.

of them being male and 18 being female, though females have a bigger ratio in this bracket (75% of all respondents), the second most relevant bracket is between 26 and 40 years old with 46 people, 41 of them male and 5 female. The bracket under the age on 18 is left with 2 male respondents and the bracket between 40 and 65 years old has 1 person of each gender. Off the 118 respondents none were above 65 years of age.

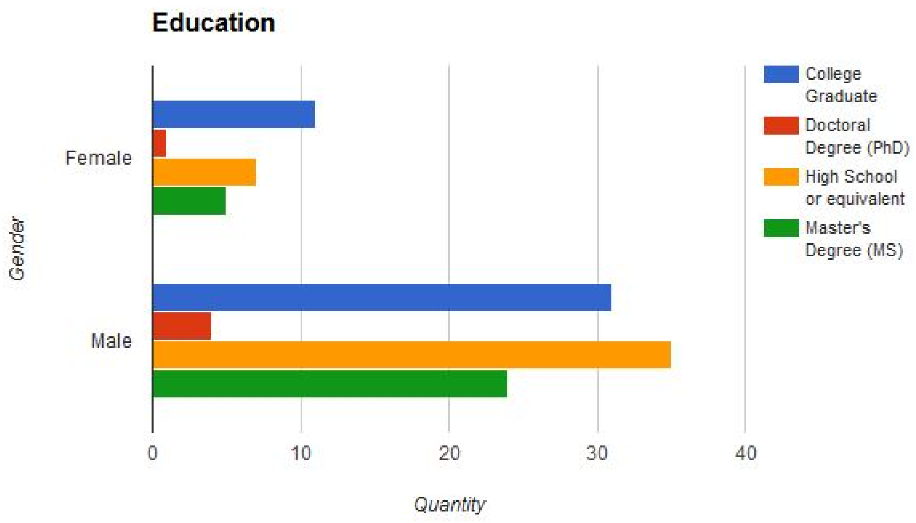


Figure 5.4: Education Level by Gender.

From the population 42 people had High School or equivalent level of education (35,3%), 42 had College Graduate level (35,3%), 30 had a Master's Degree level (25,2%) and 5 had Doctoral Degree level (4,2%). Fig. 5.4 shows education level by gender, the ratios between the genders are similar, being the only major change the one between College and High School levels, where females have a higher ratio of College level respondents (45,83%) and lower ratio of High School level respondents (29,16%), while male respondents have 26,26% and 29,66% respectively. Meaning that female respondents have a higher level education than males in this survey. Meanwhile the ratios for Master's degree is 25,53% for males and 20,83% for females, and the ratio for PhDs stands on 4,23% for males and 4,16% for females.

There was one odd data point, where the answer given was "University" and it was interpreted as "College Graduate".

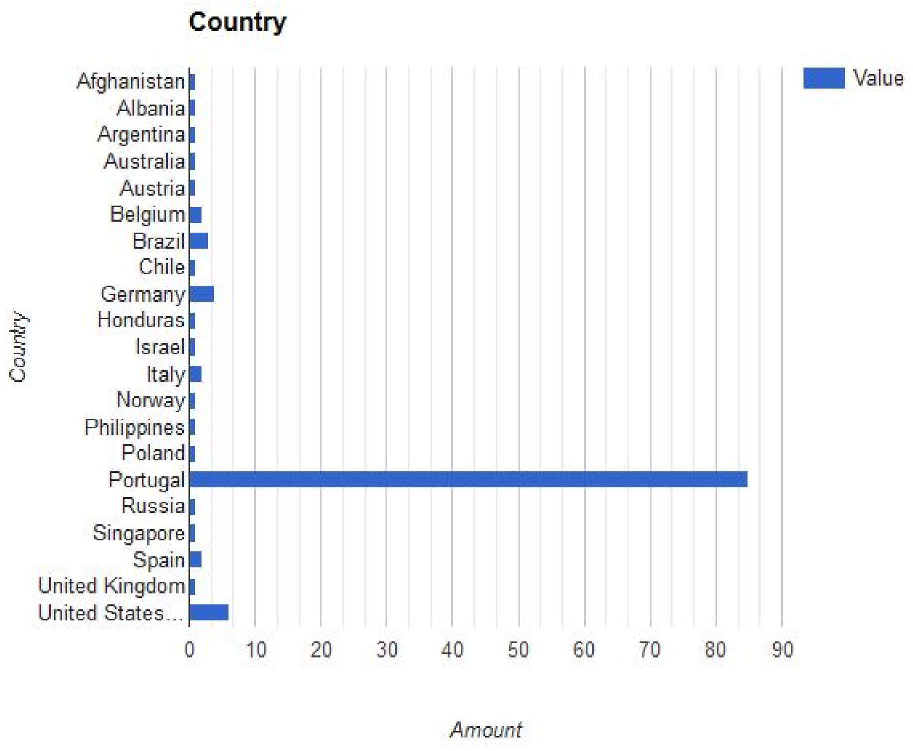


Figure 5.5: Country of the Respondents.

As expected from the way the survey was shared, the majority of the respondents are from Portugal with 85 people, 72% of all the respondents. For better visibility fig. 5.6 shows all respondents by country excluding Portugal so the data is easier to overview. International respondents are very scattered and in low number, countries with more respondents are the United States of America (6), Germany (4) and Brazil (3).

In this survey two definitions are used that might seem awkward at first, Streaming in this survey means the consumption of content that is being constantly being received and presented to the end user, this also includes Live Streaming content, but is not limited only by that, also including the consumption pre-recorded or on-demand videos like Youtube videos or more long format content delivered through a content network like Crunchyroll or Netflix. The term P2P Content in this case is regarded as the opposite of streaming, in this case the consumption of content after all files have been downloaded, where only P2P sources of file downloads are being considered. There is of course a thin line between the 2 definitions, being that for example some BitTorrent solutions offer direct streaming of content from the

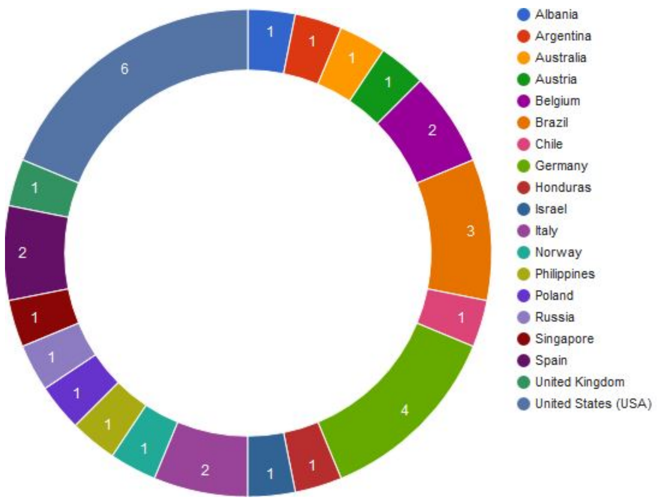


Figure 5.6: Country of the Respondents (excluding Portugal).

P2P network mashing the two definitions into an abstraction, or that streaming sites might offer the content to be downloaded and watched offline, however such considerations were ignored.

Given this, the "TV, Radio, etc" is meant to be interpreted as more traditional forms of media consumption, it was also possible for the respondent to interpret this option as the consumption of more traditional media in online form (Live Stream in case of TV Channels or Podcasts/Live Radio in case of Radio Channels). Originally the question only had the first two possible answers, but as a multiple answer question a new perspective could be had by having this option, not having any detriment on the quality of the data in my opinion.

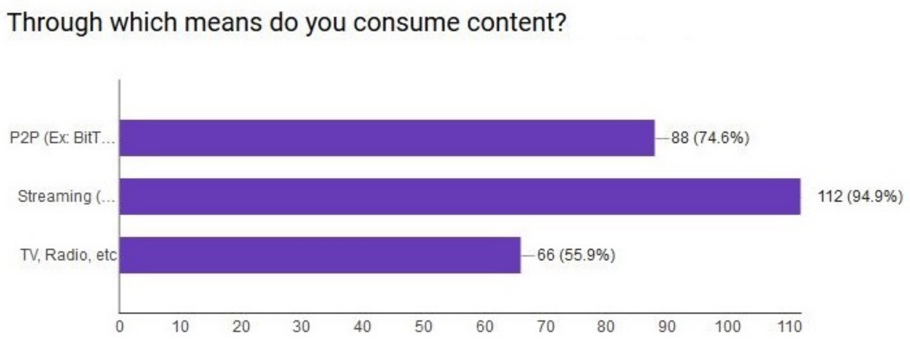
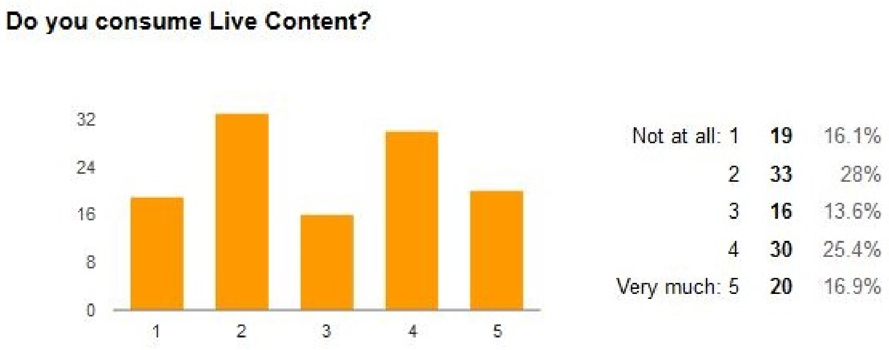


Figure 5.7: Type of Content Consumption.

A large majority of the respondents reported consuming Streaming content (94%) while 74,6% reported consuming P2P content. 70,33% of the respondents consume P2P and Streaming contents, 44,91% consume all 3 types of content and 44,06% don't consume TV or Radio at all, while only 5,08% (6 respondents) said they only consumed one type exclusively, of those respondents 66,66% consumed only P2P (4 respondents). Of the total population only 5,08% (6 respondents) reported not using Streaming at all.



Not at all: 1	**19**	16.1%
2	**33**	28%
3	**16**	13.6%
4	**30**	25.4%
Very much: 5	**20**	16.9%

Figure 5.8: Consumption of Live Content.

The consumption of live content is rather balanced across respondents, around the same amount of respondents are among the groups that consume more live content and not at all (~16%), the moderate consumption of live content is also close to that range, with 13,6% of the respondents. The rest, being where most of the respondents lie, stay in between the neutral and lower/higher levels of consumption with 28% and 25,4% respectively.

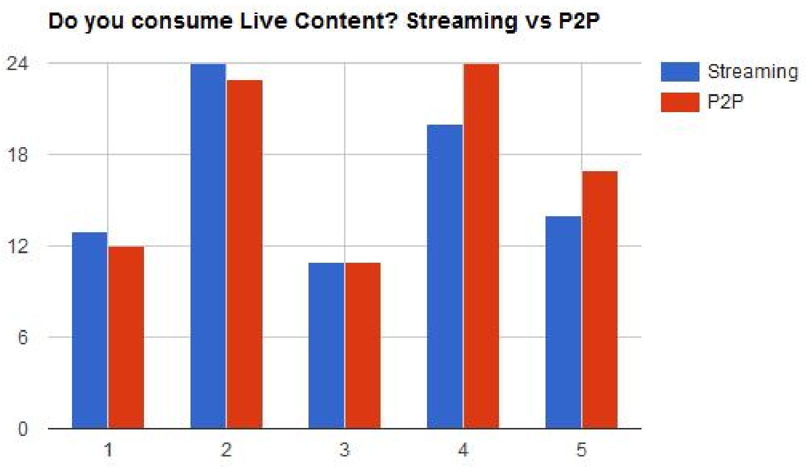


Figure 5.9: Consumption of Live Content from Respondents that consume Streaming vs P2P.

It was intended to assert from the data, the difference in consumption of respondents that usually consume P2P against respondents that consume Streaming. However as it can be seen in fig. 5.9, there is not much difference between the two groups, even more strangely, respondents that consume P2P revealed to consume more live content than respondents that reported to report Streaming. As Live Content is included in Streaming and the respondents that consume P2P that consume Live Content also, inherently consume Streaming content.

No relevant conclusion can be made from this data, this probably has to do with most respondent (70,33%) being consumers of both Streaming and P2P. However, respondents that did not consume P2P did tend to a lower consumption of Live Content.

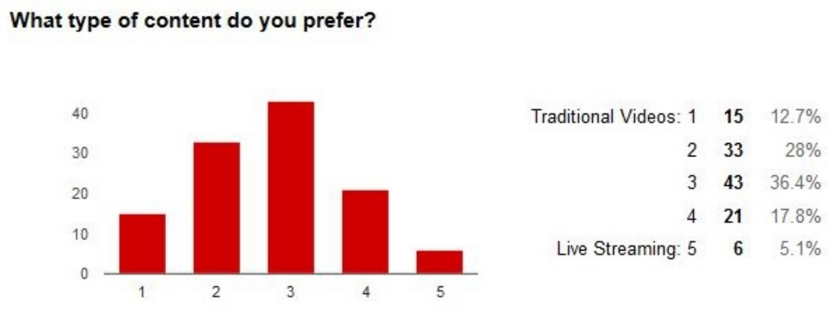


Figure 5.10: Prefered content, Traditional vedios vs Live Streaming.

40,7% of the population tend for traditional videos, 36,4% remain neutral and the rest, 22,9% tend for live streaming. This question unlike fig. 5.8, aims to understand preference in regard to content disregarding the amount of consumption. It is clear that respondents tend to prefer traditional videos over live streaming content. Respondents that consume streaming tend to traditional videos, even when they also consume P2P and TV, Radio content, Meanwhile users that exclusively consume P2P or TV, Radio remain completely neutral in preference for the two.

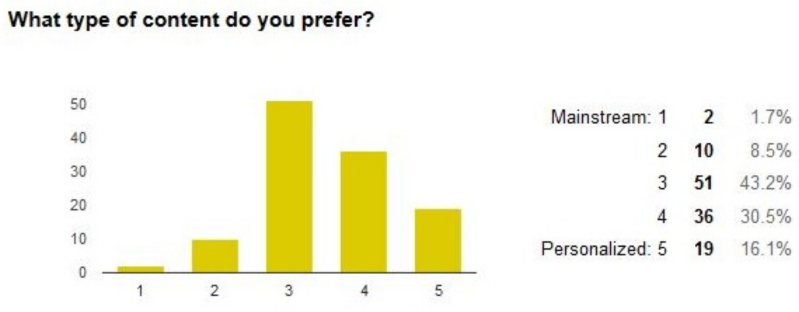
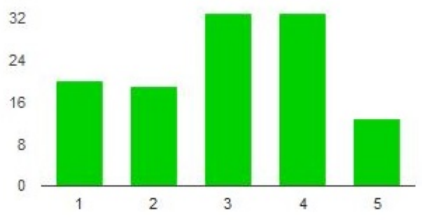


Figure 5.11: Prefered content, Mainstream vs Personalized.

10,2% of the respondents tend for mainstream content, 43,2% remain neutral and 46,6% prefer personalized content. Respondents that chose to be neutral between the two options are in a higher number in this question when compared to fig. 5.10. Almost half of the population tended towards the preference of personalized content among all groups, being that respondents that consumed only P2P or only TV, Radio tended to neutral. Respondents that consumed more live content (fig. 5.8) had a higher tendency for personalised content than the average.

On the curious side, fig. 5.10 and fig. 5.11 show a totally different pattern, though Live Streaming contents being usually regarded to being more personalized content than traditional videos.

Do you consider that social tools enhance the content you consume?

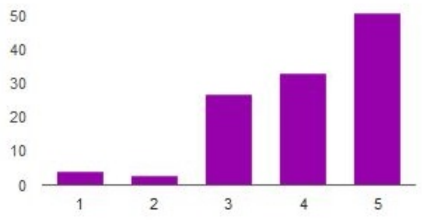


Not at All: 1	**20**	16.9%
2	**19**	16.1%
3	**33**	28%
4	**33**	28%
Very much: 5	**13**	11%

Figure 5.12: Social Tool contribution for content enhancement.

39% of the respondents tend towards Social Tools enhancing the consumption of content, 33% of the respondents think otherwise, while the rest 28% remain neutral. There is a slight inclination for social tools to be considered to enhance the consumption of media, respondents that said to prefer personalized content tended to consider that social tools were relevant, but less so than other respondents. Respondents that prefered mainstream content (10,2%) tended to consider social tools unimportant.

Is privacy while consuming content important to you?

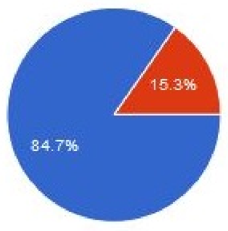


Not at All: 1	**4**	3.4%
2	**3**	2.5%
3	**27**	22.9%
4	**33**	28%
Very much: 5	**51**	43.2%

Figure 5.13: Importance of Privacy.

The majority 71,2% of the respondents consider privacy important when consuming content, 43,2% consider it very important. 22,9% are neutral and only 5,9% regard it as not an important matter.

Have you ever noticed content being deleted or censored?



Yes	**100**	84.7%
No	**18**	15.3%

Figure 5.14: Notice of Content Censorship.

84,7% of the respondents answered that they have noticed content being deleted or censored while consuming media, while 18 (15,3%) said they didn't.

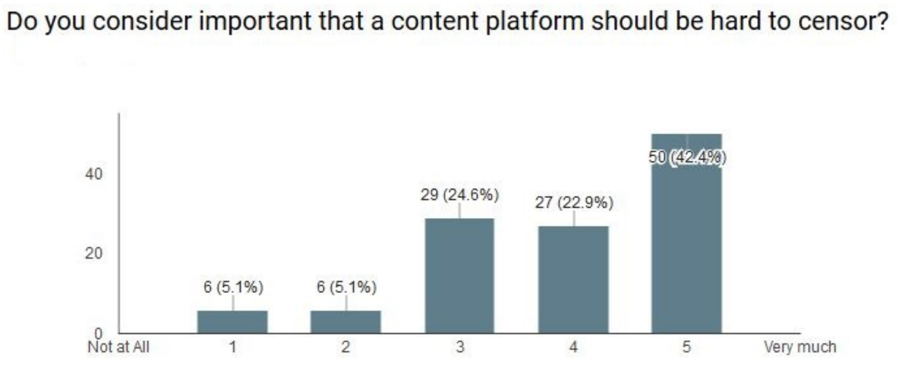


Figure 5.15: Importance of censorship resilience.

5,3% of the respondents consider that a content platform should be hard to censor, 24,6% remain neutral while only 10,2% consider it not important.

Those than answered "no" in fig. 5.14 remained mostly neutral in this question, asserting that respondent that don't notice any censorship remain neutral in regard to its relevance while consuming content.

In conclusion, when looking at this data and comparing it with concepts that Seedseer tries to establish, some points come in favor of the solution. Traditional videos being more relevant over Streaming go in favor of static content in P2P, one of the strong points of using hashed content being that it becomes easier to proliferate. Seedseer being community-centered is a factor that is hard to take as a hard advantage, data shows a preference for personalized content which would go in favor of it, but social tools while deemed relevant, its relevance wasn't overwhelming enough for me to conclude it as a clear advantage. While Streaming content was used by 94,9% of the users, live streaming isn't as relevant as one would expect. This is a functionality that was not considered for seedseer and the data above suggests it isn't important to most users. Concerns for lack of privacy and for censorship were highly regarded by respondents, while seedseer had these in mind ever since it's development days and tackles the issue multiple times, a serious comparison would have to be done as to whether it would increase the privacy when compared to similar projects in the market. The current WebRTC information exposure in the protocol is the main point for privacy concerns in seedseer. On the other hand, seedseer definitely reduces the risk of censorship which can be taken as an advantage.

This survey did not tackle the respondents opinion on performance for consuming content or ease of use of P2P software. One of the major trade-offs of seedseer is that performance-wise it is inherently worse than other solutions, both for being more decentralized and not a native application, among other reasons explained in previous chapters. Whether a user would willingly make this trade-off is hard to say, however the large majority of the users do have privacy concerns. As for ease of use, it is one of the advantages of seedseer but no data was gathered from the survey regarding its importance for users. Although it can be asserted from the data that the majority of surveyed users do use streaming websites more than P2P software, which could be argued that it's ease of use is indeed a relevant factor for them.

5.2 Performance, Limitations and Tests

On this chapter several tests will be performed on how the prototype behaved in specific situations, some tests will be performed using external tools to determine limitations of the prototype in an effort to get to correct assertions on why they exist.

For the tests realized in this chapter the following setup was used:

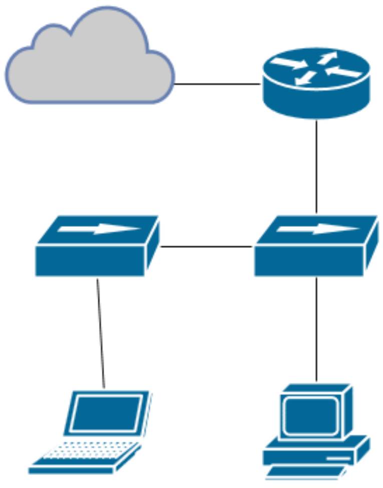


Figure 5.16: Network Diagram of the setup used for testing.

The setup used for tests in this chapter is shown above in fig. 5.16. In this intranet two computers will be used, a laptop and a desktop that are in different locations and connected via 100Mbps Hubs. One of these hubs is connected to a router that gives access to the Internet through cable connection with 200Mbps downstream and 20Mbps upstream. The laptop runs a Supernode that is accessible both in the Intranet and the Internet, the supernode runs on default on TCP port 80 with port forwarding and has a Dynamic DNS (DynDNS) configured on the router for easier access from the Internet. Unless specified, one or multiple peer clients run on the desktop, these are connected to the Supernode that is set up on the laptop.

To better interpret the data that follows it is relevant to keep in mind the hardware that was used and it's possible limitations, some of these specifications are shown below:

Laptop - Supernode
– CPU Intel i7-4700MQ 2.40GHz
– RAM 16GBs
– Intranet speed 100Mbps
Desktop - Peers
– CPU Intel Quad Core Q6600 2.40GHz
– RAM 2GBs
– Intranet speed 100Mbps
Internet speed
–200 Mbps Download
– 20 Mbps Upload

Listing 6: Test Bed Specifications

Notice that the Supernode running on the laptop is limited to 100Mbps for both download and upload while in the intranet but only 100Mbps of download and 20Mbps of upload to the Internet. Peers on the desktop have the same exact limits although the 20Mbps of upload are shared, both the laptop and the desktop can download at 100Mbps at the same time.

Client Load

Supernodes have the capability of giving all the code needed for regular peer, though as stated in the previous chapter this isn't required and peers can simply establish the WebSockets connection to a supernode. The first results are tests on the supernode capability to deliver all files needed for the client to run. This test will have 6 different metrics, total time, average call time, slowest call time, DOM content loading time, DOM processing time and time to first byte. The total time is the time from the moment the request is made by the browser to the moment the DOM finishes processing. The slowest call time is how long the slowest resource request took.The average call time is the average time of all the resources requested by the page. DOM content loading time is the time that event took to execute after firing, this is contained in the DOM processing time, that is the time since the HTML is received until the page is fully loaded. Lastly, the time to first byte is the time from when the request started until the first byte of the webpage arrives, this metric is often relevant to identify how reactive the server is, but doesn't necessarily affect the final load time, as a server can start sending the header fast but take a long time to finish the request.

The peer client constitutes a 78KB transfer of 11 requests, 10 of them are handled by the supernode and one external call to a JS file that is used to retrieve the peer's geolocation. All of the tests were performed with caching disabled so it wouldn't affect the load times positively.

The fig. 5.17 is made of 8 client loads executed in the same computer as the supernode. The slowest call is the JS request on the external site that takes around 400ms while the average requests take less than 100ms. The total load times are around 450ms, which is

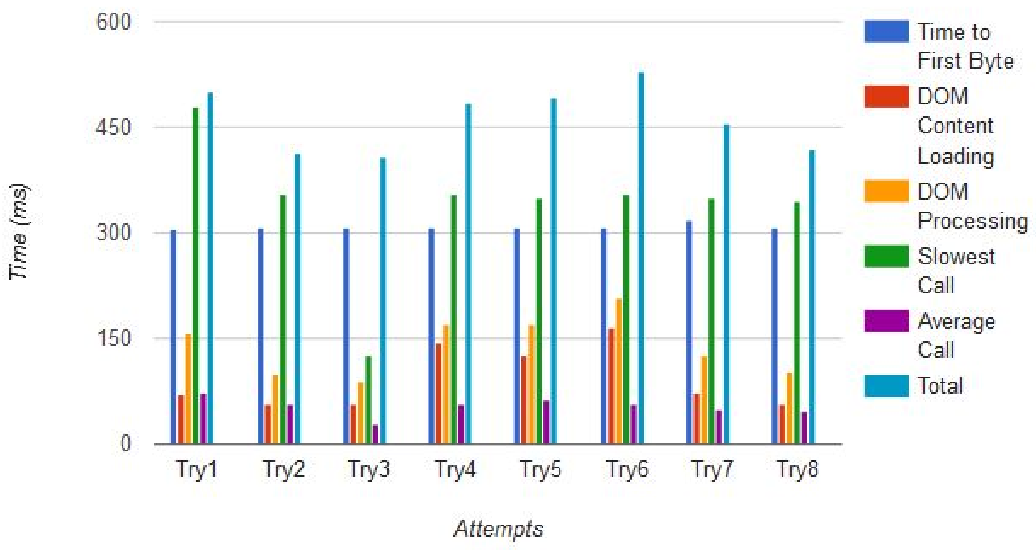


Figure 5.17: Local client loading of peers.

acceptable, having in mind that the time for the first byte is around 300ms. By the end of the total load time everything has finished loading except the geolocation request, most of the requests and rendering in the browser happen in parallel, therefore the delays don't add up to the total time. The total time does however include the websocket connection as well as receiving the state of the supernode that is represented by the welcome message, full userlist and downloads list (if anon mode isn't enabled). Client loading times can be optimized if anon mode is activated and geolocation is disabled. Compression is also an option that would allow better loading times while sacrificing the time to the first byte by the time taken to compress.

All javascript code is already minified for regular peers, but as all content provided by the supernode is static and all the dynamic data comes after the client load by websocket, pre-compression is a viable option if the client load reveals to be a bottleneck at any point.

Fig. 5.18 represents the same test as fig. 5.17, but this time the test is run on the desktop that is on the same network. Strangely the delay significantly lower than the previous test that was run on the local machine where the supernode was running. The difference comes from the time to the first byte that is 110ms on average, almost 3 times faster than the previous test, affecting the total time that averages at 380ms, 70ms less than the test ran in the laptop. A possible reason for this is the high number of processes running on the laptop while the tests were running that might have affected the time for the first byte on the local chrome instance. The difference in performance from the laptop and desktop can be seen clearly in the difference of execution times on the DOM Processing, being that on this test the desktop took over 200ms on average while the laptop's average lies below 150ms. In this test as the load times were lower, which made the geolocation request actually take more

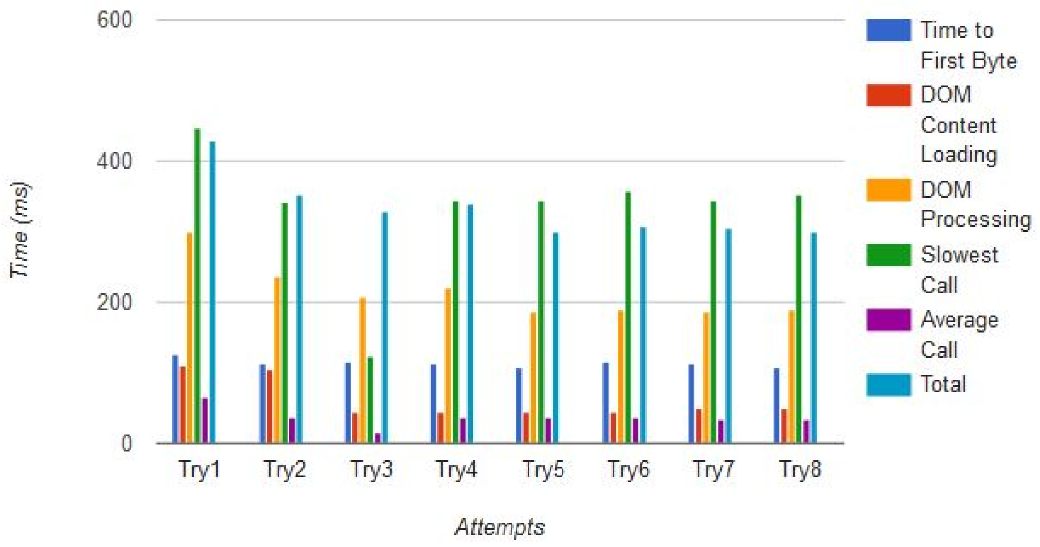


Figure 5.18: Client loading of peers.

time than the time to fully load the client on most of the tries.

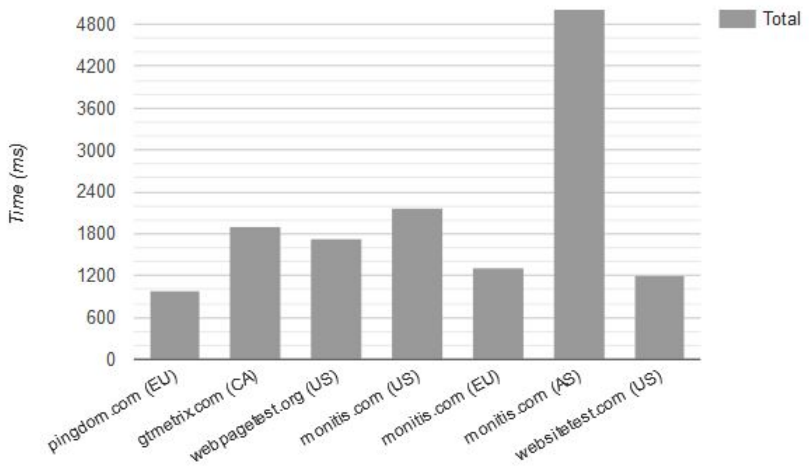


Figure 5.19: Client loading of peers by Internet page loaders.

To test client load from the internet several pageloading sites were used. [2] [3] [4] [5] All these sites do a full load of the client, which is relevant to test the client loading, sites that ignored DOM Processing or didn't establish a websocket connection weren't considered. The results in fig. 5.19 reveal around 1 to 1.3 second of load time for Europe locations as well as for

[2]https://tools.pingdom.com/#!/d6nKkt/http://seedseer.no-ip.org/
[3]https://gtmetrix.com/reports/seedseer.no-ip.org/VcYeUp97
[4]http://www.webpagetest.org/result/160601_89_VNE/
[5]http://websitetest.com/ui/tests/574e72a46ac6c69c01000005

"websitetest.com" that is located in the US. The other locations in the US and CA obtained load times near the 2 seconds range, the Asia location got the worst result at 5 seconds load time.

For a javascript HTTP server running on a normal instance of chrome, these ranges of load times are good and are close to what a normal HTTP server would achieve, specially considering that all these tests also connected through websockets and got all the initial data from the supernode.

Transfers

In this subchapter tests will be made to transfers using both BitTorrent and WebRTC. All WebRTC tests were performed on the desktop with multiple peers, the BitTorrent tests were all performed on the laptop, as the supernode is the only client that supports it. All images shown are from the receiver's point of view.

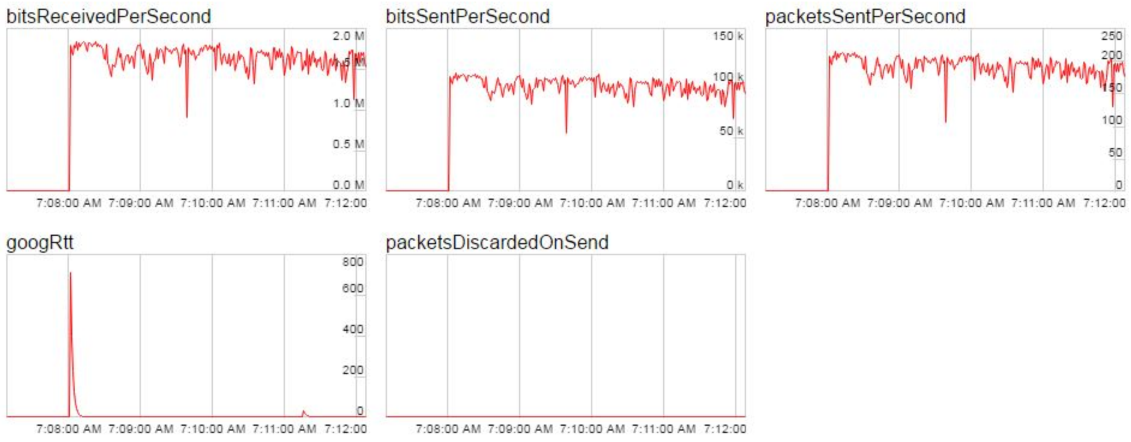


Figure 5.20: WebRTC Transfer with a 512 bytes chuck size on UDP.

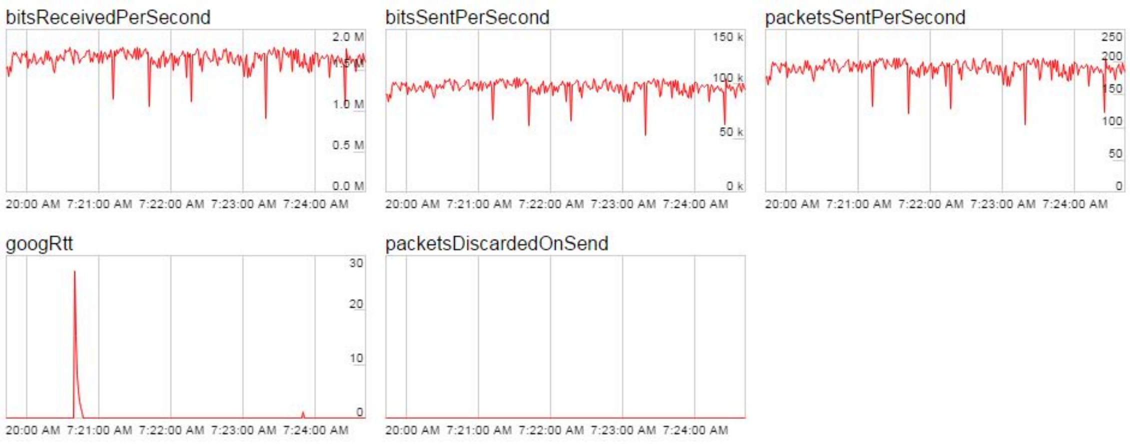


Figure 5.21: WebRTC Transfer with a 512 bytes chuck size on TCP.

Both tests shown in fig. 5.20 and fig. 5.21 were executed with the same parameters where the only change it's that the first test uses an unreliable connection over UDP and the second uses a reliable connection over TCP. The first test achieved a 215KB/s with a maximum speed hit of 263KB/s while the second achieved 208KB/s and a maximum speed of 247KB/s. After multiple tests with the same settings it can be asserted that both achieve the similar performance. The difference between the two are on packet loss situations, on TCP the packets are retransmitted automatically and the packets are ordered. For file transfers this is a trade-off where TCP uses more overhead and UDP needs checks on the application level for retransmissions. The slight difference on UDP against TCP in data transfers is that UDP is typically faster due to the reduced overhead, though on the tests performed this wasn't noticeable.

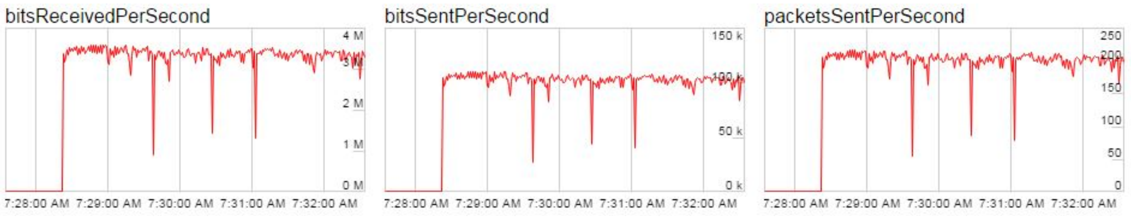


Figure 5.22: WebRTC Transfer with a 1024 bytes chuck size on TCP.

As we can see in fig. 5.22 the number of packets sent for requests didn't increase much when compared to fig. 5.21, but the download rate almost doubled. The average download speed was 421KB/s while the maximum speed achieved was 459KB/s.

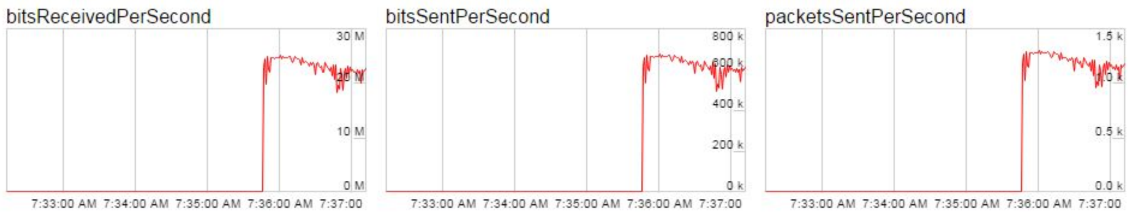


Figure 5.23: WebRTC Transfer with a 8192 bytes chuck size on TCP.

In fig. 5.23 we can see the same test as before but with a chuck with 8192 bytes of size. The graph is rather small because the file being used finished at the end of the graph. In this case the number of packets sent was 5 times higher into an average of 1200 packets per second, the average speed of this test was 2821KB/s and the maximum speed was 3285KB/s.

In fig. 5.24 it is shown the graphs for twice the previous chuck size, this time with 16384 bytes the average number of packets sent per second is 2400, achieving a maximum speed of 6244KB/s reaching half of the max bandwidth of the peer with only one seeder. The average speed was 5473KB/s managing to finish the download of the file in under a minute.

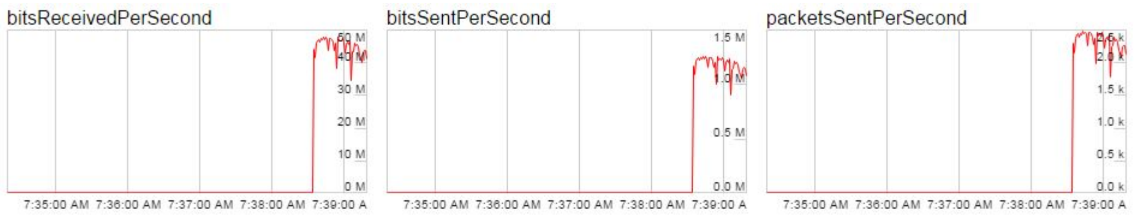


Figure 5.24: WebRTC Transfer with a 16384 bytes chuck size on TCP.

As it is discussed in many developer forums, this chuck size is the maximum size that is advised by many for current chrome implementation of data channels, given the size of chrome's buffers for WebRTC using SCTP.

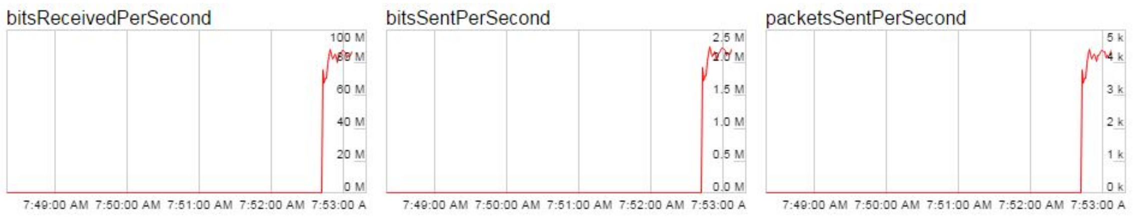


Figure 5.25: WebRTC Transfer with a 65536 bytes chuck size on TCP.

Above the advised chuck size the current seedseer's implementation experiences random crashes that stop the transfer, however fig. 5.25 shows one test where it successfully transferred the file in 24,43 seconds with an average speed of 10191KB/s and a maximum speed of 11571KB/s, achieving almost the total available bandwidth of the peer with only one seeder.

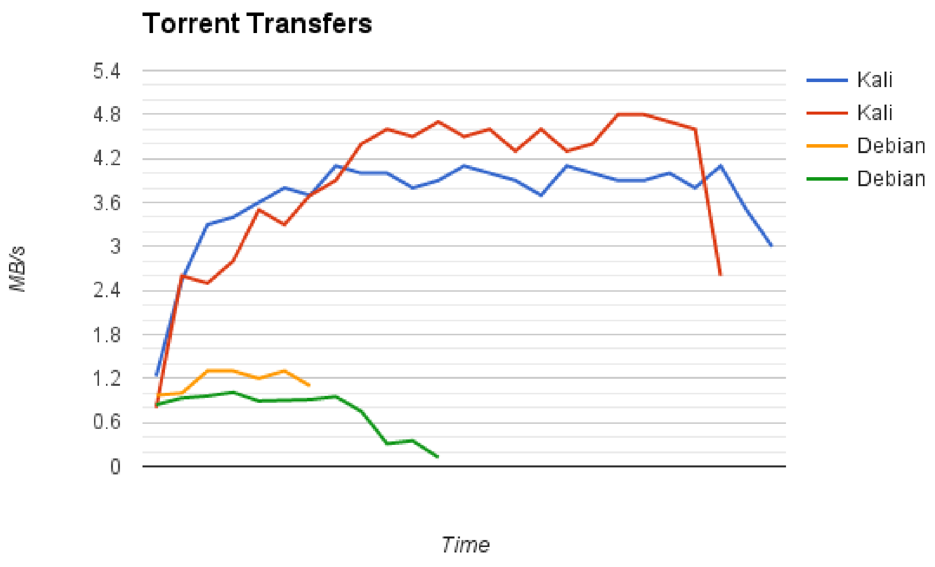


Figure 5.26: BitTorrent transfers of 4 linux distros.

Fig. 5.26 shows a test of downloading the same 2 torrents twice, one of a kali linux distro and another of a light debian linux distro. As seen in fig. 5.26, the BT transfers have a slow start and a slow endgame, both are to be expected due to the way the bittorrent protocol works. In the example above no trackers were used, so finding peers with the DHT takes a bit longer than it would simply with the usage of trackers. While active connections mature the speed goes up and remains somewhat constant until endgame mode.

While these speeds are below the WebRTC implementation it is relevant to have in mind that the BT implementation uses tit-for-tat and is competing for resources with other peers, while the WebRTC implementation is rather simpler and more willing to share resources without rewards. The average upload speed on these tests was 4KB/s, which might have discouraged other peers to increase the client's speed, despite most of the swarm being seeders.

Supernode limitations

For the following tests a tool called Webserver Stress Test 8 was used, in the desktop machine the tool is limited to 4000 virtual concurrent users. While the objective of this tool is to test the limits of the supernode, it is pertinent to make sure the machine running the stress test software isn't getting bottlenecked instead of the supernode being tested. The tests were set so the desktop machine wouldn't bottleneck on CPU performance or on trying to open sockets, producing bad data just due to these limitations. This will test the capability of the server to send the the peer code, the tool doesn't emulate the browser so the results won't evaluate DOM Processing, run javascript or connect through websockets to the supernode. However the tool provides useful data to understand possible limitations on the Supernode while handling a large number of requests, clients and packet output.

Ramp Tests

A ramp test constitutes in a gradual increase of users over time until the set max number. One minute before the test ends, the maximum number of users is reached and is maintained for the last minute until the test ends. This type of test escalates the number of users steadily in order to determine the maximum number of users the server can handle. By increasing it slowly the goal of the test is to identify how many users the server can handle before producing error messages.

In this test a supernode will be tested. Every active client requests the root page from the supernode, that basically serves a copy of a peer client. Each client repeats this every X number of seconds until the test is finished. In this case the amount chosen is 6 seconds. Figures 5.27, 5.28 and 5.29 represent the first ramp test that was performed with 3400 users.

In this case the test was performed in 10 minutes with no errors, as seen in fig. 5.27 near the 5 minute mark some users experienced up to 10 seconds of delay, but most of them remained unaffected. Fig. 5.28 shows the average delay of all active client requests, some delay spikes can be seen after the 2 minute mark, placing the maximum average request time

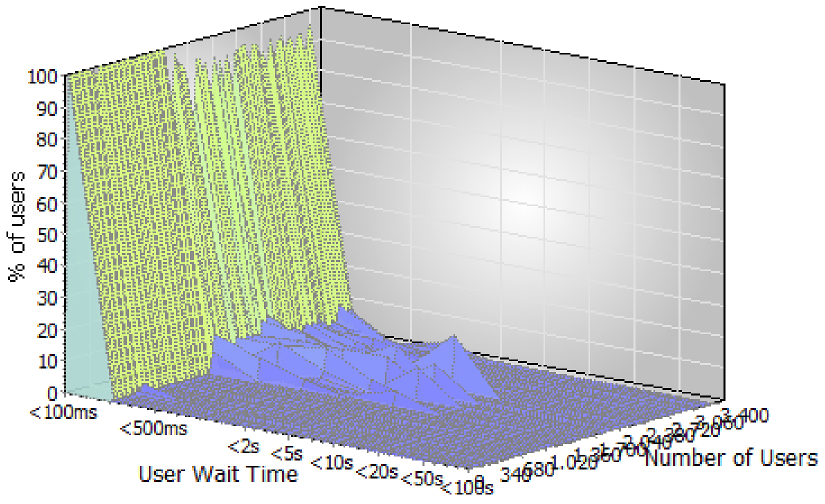


Figure 5.27: Spectrum of load times of the Ramp test with 3400 users.

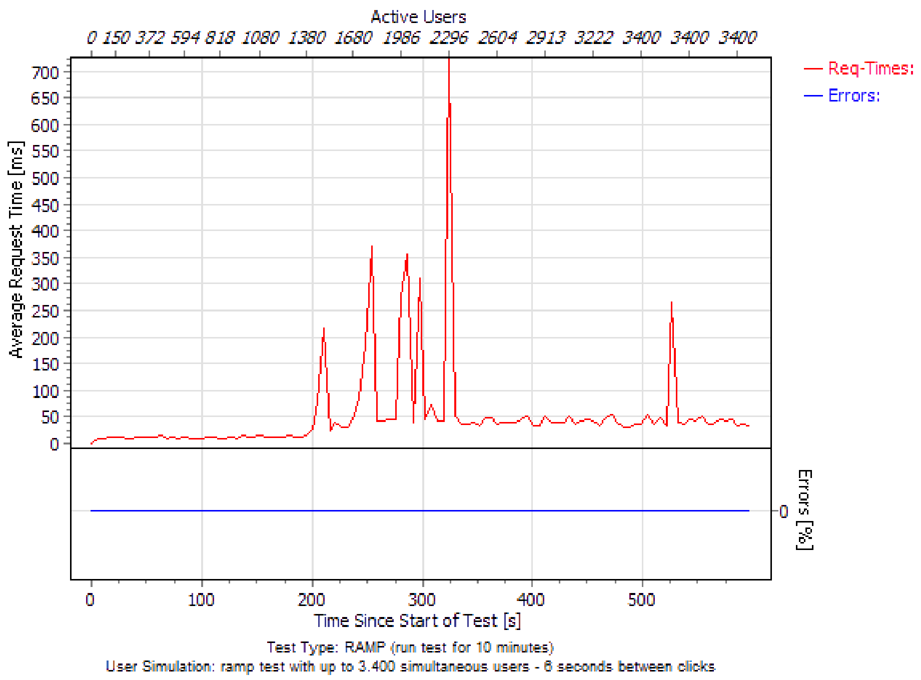


Figure 5.28: Load times and Errors of the Ramp test with 3400 users.

of this test around 700ms. Disregarding the delay spikes, the average request time scales overtime as more clients join in gradually, the initial average time goes from around 15ms on the initial requests to around 45ms when all the 3400 clients are active. As in this test every client does a request every 6 seconds, this leaves the test theoretically at about 34.000 requests per minute, or 2.040.000 requests per hour, actual log data points to 278 requests per second, or 1.002.001 requests per hour which is an acceptable number of requests for the

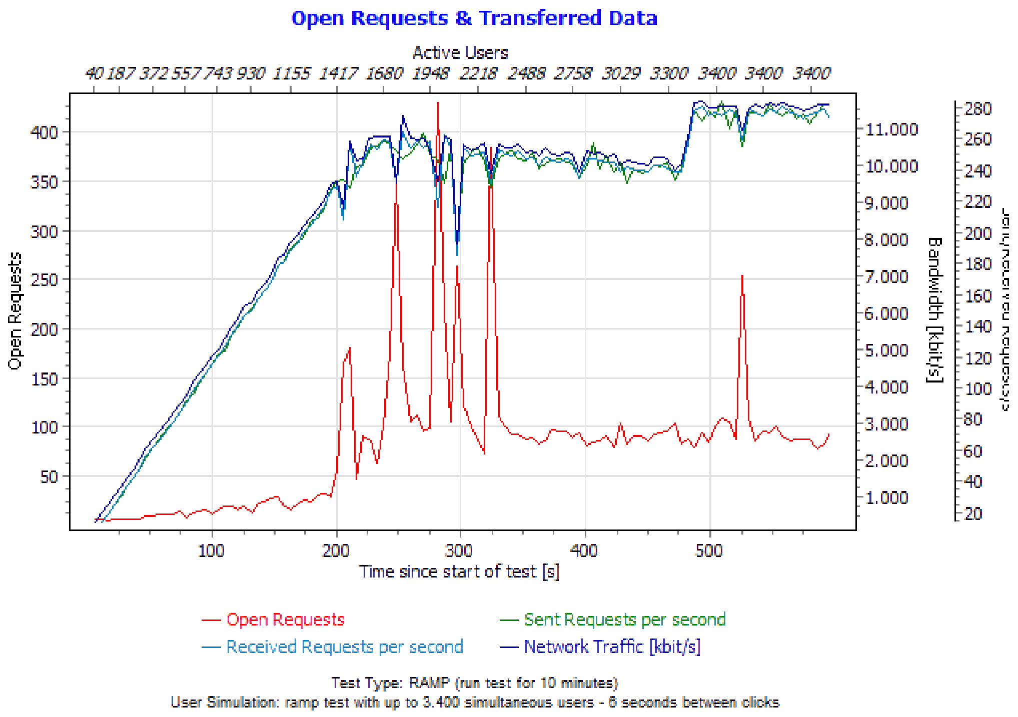


Figure 5.29: Open requests and Transferred Data of the Ramp test with 3400 users.

average 61ms reply in this ramp up test.

The spikes in the middle of the test seem abnormal, probably due to the fast increase in bandwidth and open requests piling up, this happens between 200 to 330 seconds into the test where open requests spike to 400 while bandwidth has downward spikes and load times increase among a considerable amount of users. The worst data point is at 327 seconds with 17,42% of the users taking between 3 to 5 seconds to load, however after that the data normalizes back into 97% of the users being under 200ms of load time.

This test represents a ramp test with 4000 users in 15 minutes, where each user requested a page load every 6 seconds. As we can see near the end of the test, in fig 5.32, the test machine got bottlenecked waiting for local sockets, though a spike can be seen at second 730, after the first 5 minute mark and with around 1500 active users the delay for acquiring local sockets is already above the recommended 100ms threshold, this explains the bandwidth spike at 730 seconds when the test reaches it's maximum set number of users and the test machine is finally able to load the remaining virtual users. A similar pattern can be seen in the previous test 1.

While the theoretical number of requests is higher than the previous test, the actual number of requests is around 274 per second, or 986.784 per hour which leaves it a bit lower than the first test despite having 600 more simultaneous users. The data compared to the previous test is also better, less load times for most users, a lower average delay of 33ms and less simultaneous open requests, one possible explanation for this is in the extra 5 minutes this test had to ramp up when compared to the first, being that both the supernode and the

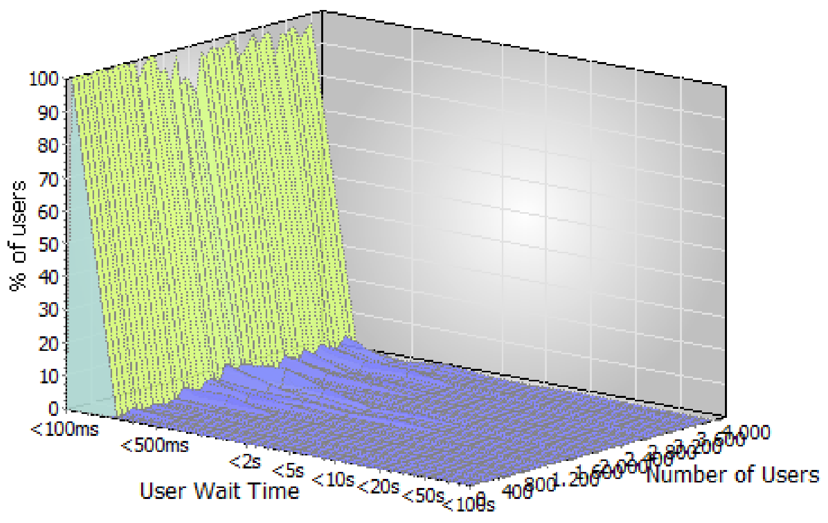


Figure 5.30: Spectrum of load times of the Ramp test with 4000 users.

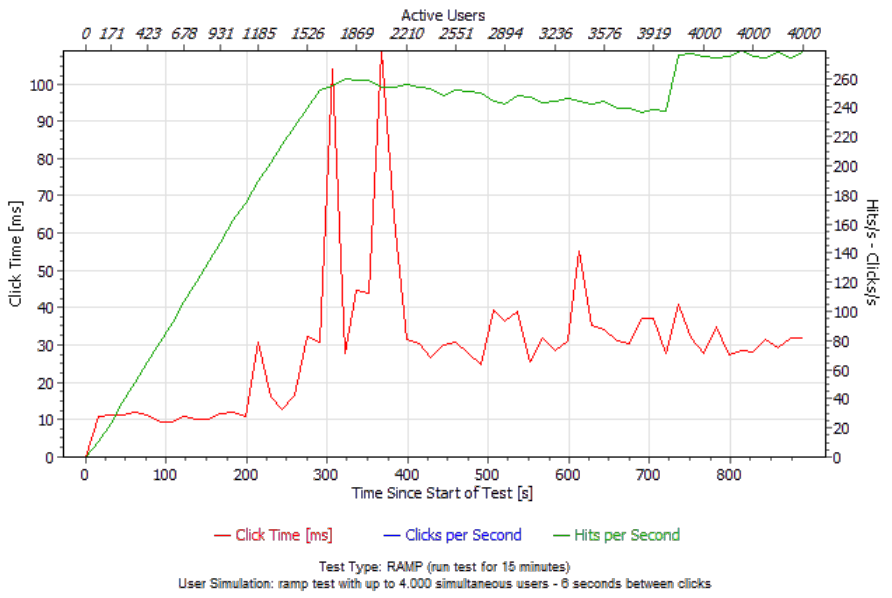


Figure 5.31: Load times and hits per second of the Ramp test with 4000 users.

test machine had less stress and more time to react to the gradual load.

From the supernode point of view, the two tests induced a similar result in resources used when the tests reached the max number of users. RAM-wise there were no relevant changes while the tests were being performed, as for CPU usage it gauged between 7% and 14%, being constant at 11% usage throughout most of the time.

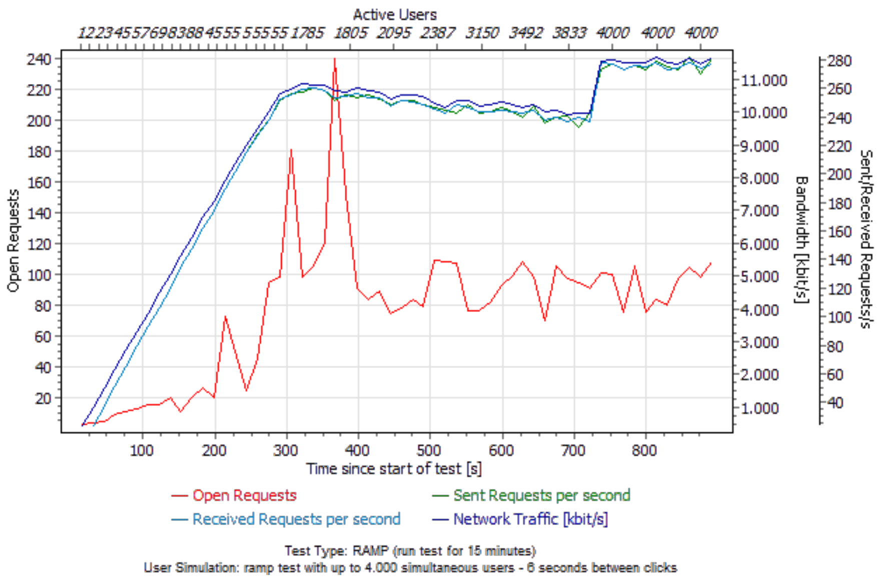


Figure 5.32: Open requests and Transferred Data of the Ramp test with 4000 users.

Load Test

A Load Test constitutes in a test where a constant load of users is maintained continuously for all the testing time. The goal of this type of test is to check if the server is able to serve all users while being under constant load. Typically the assigned number of users is set to an expected level or above.

This test is a constant load test, meaning that unlike a ramp up test where the users are initiated gradually, in this test all users are active since the start and remain active until the end of the set time. This was performed with 3400 users in 15 minutes, where each user made a request every 6 seconds leaving the theoretical request volume at 34.000 requests per minute, or 2.040.000 requests per hour, the same load as the first ramp test.

The test machine took 57 seconds to start all users, after that as shown in fig. 5.34 it keeps a constant load a little above 280 page loads per second, averaging at 276 per second due to down spikes. The test starts with high bandwidth usage and remains constant throughout the test. Due to a little higher number of requests being made, in average the throughput is 11.749 kbit/sec, a bit higher than the previous two ramp tests. As for simultaneously open requests as shown in fig. 5.35, the results seem similar to the previous tests but with a lot more spikes reaching 200 to 250 simultaneous open requests. The load time average stays at 58ms, while in fig. 5.34 the load times on average are higher but don't look much different than the ramp tests 5.285.31. In fig. 5.33 it clearly shows that a considerable amount of users experienced high load times when compared to fig. 5.30 from the second ramp test, despite that thest having more simultaneous users.

Of all the 232.561 requests, 74 of those were errors, being the only test where the supernode failed to reply, 27 of those occurred on second 253, 1 on second 297, 1 on second 581 and the remaining 45 errors occurred on second 887. Looking in detail, the first 27 errors and the

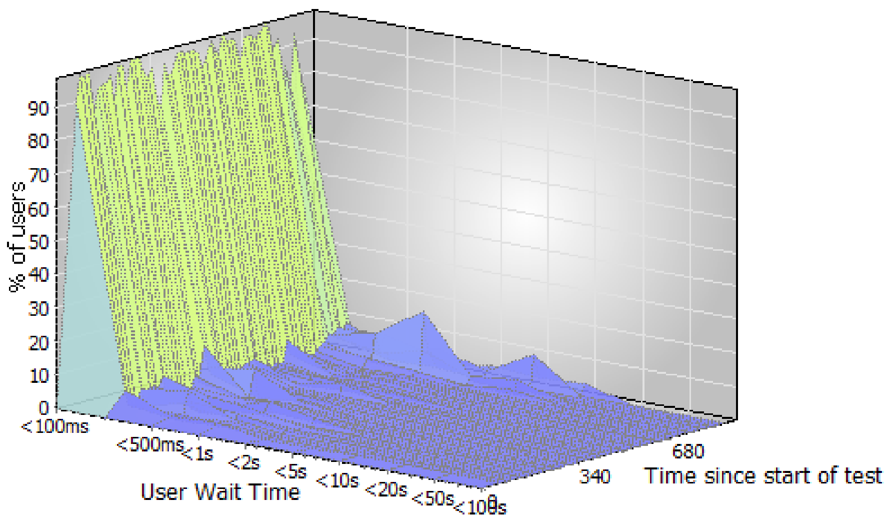


Figure 5.33: Spectrum of load times of the Load Test with 3400 users.

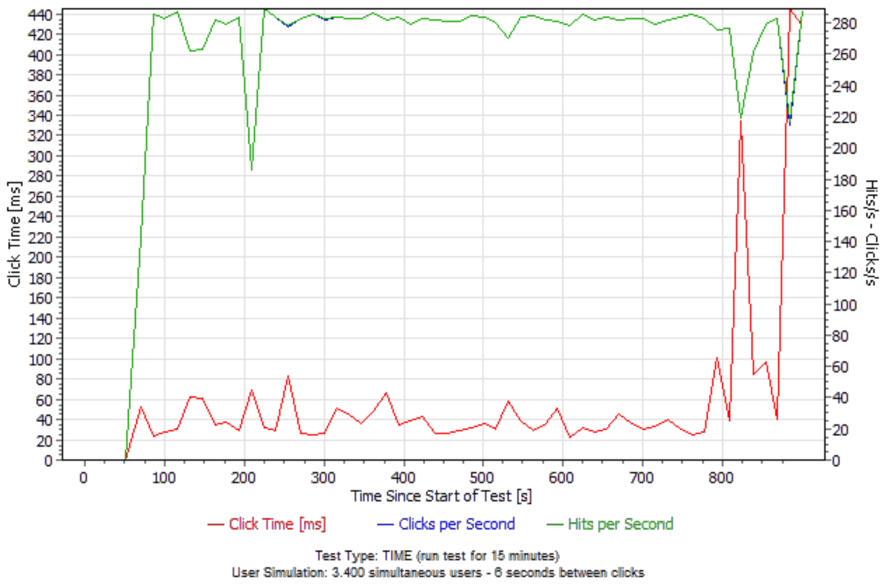


Figure 5.34: Load times and hits per second of the Load Test with 3400 users.

one at 581 seconds were due to the connection being refused, the one on second 297 reported an invalid host name and the remaining 45 errors reported an unknown http result. While the last 45 errors are understandable due to the clients being in stress with a high spike in requests, delay and bandwidth, it is unclear from the data if the other errors were due to a limitation on the supernode or on the test machine. When compared to the total number of requests, the errors don't have much relevance.

Other load tests were performed to force errors in order to find possible bottlenecks, however due to limits of the setup, the data was unreliable as a result of the testing machine

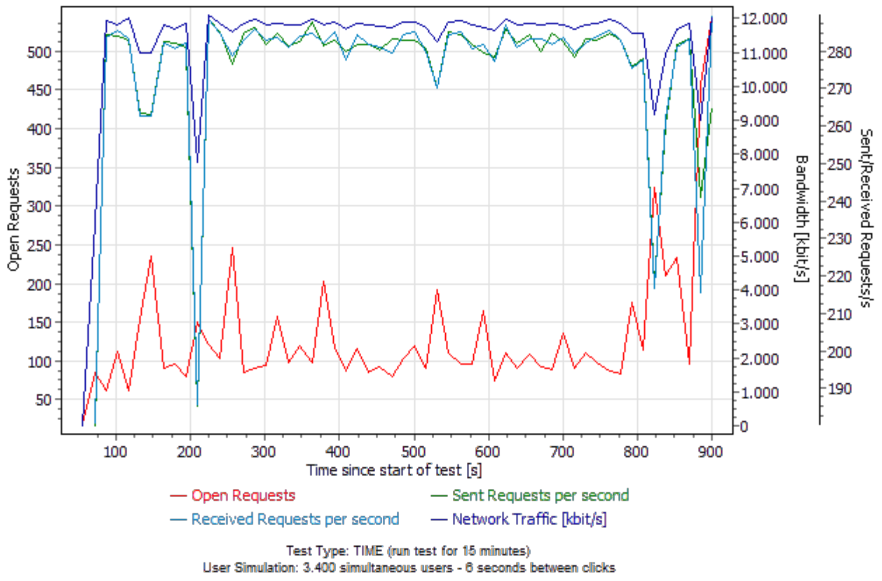


Figure 5.35: Open requests and Transferred Data of the Load Test with 3400 users.

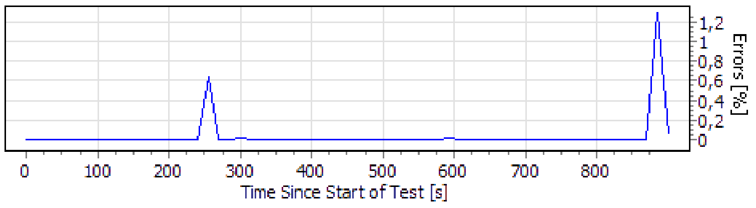


Figure 5.36: Error Rate on the Load Test with 3400 users.

being bottlenecked on its resources before the supernode.

Similarly to the ramp tests, the usage of resources remained low. There was no relevant changes in RAM usage, and CPU usage gauged between 8% and 14%, averaging at 12%. As the content being served in the tests is static content and really small in size when compared to a typical website, there is not much bandwidth usage considering the amount of users doing requests. Though the tests above didn't reach the point of a bottleneck, the supernode shows to be capable to handle a high number of users and requests.

Bandwidth Test

A Bandwidth test consists in maxing the throughput of the server, in this case the supernode, in order to perceive its max bandwidth and how the high use of bandwidth affects the use of other resources.

To perform this test, it was set as a ramp test with gradually increasing users, where 8 users download a file from the supernode through HTTP. In this test each user downloads the file every 7 seconds. The file chosen was a video file, an mp4 that has 11,4MB (12.006.066 bytes). The download rate is shown in fig. 5.37, but it can also be calculated by dividing the size of the downloaded file by the time used to download the file.

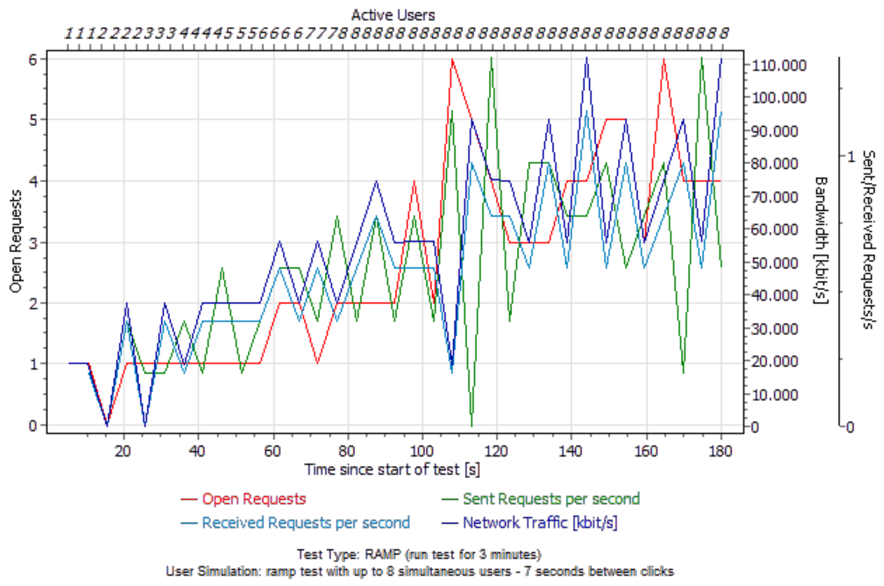


Figure 5.37: Transferred Data of the Bandwidth Test.

As seen in fig. 5.37, the first users downloaded the files without competing for bandwidth, meaning that they finished the download in less than 2 seconds. Gradually as users increase, the bandwidth is divided between the users actively downloading at the time. This results in the supernode reaching it's maximum bandwidth, which consequently increases the average download times for the users, as the bandwidth is divided between them. At 80 seconds into the test, all the 8 users are active, but simultaneous downloads only start to pile up 20 seconds later. In the last minute of the test there are always open requests and the server is in constant load to deliever them. While the values in the figure are averages, two of the points in the graph are near the maximum speed that the supernode can achieve.

	Avg. DL Time	Requested	Finished	Errors	Downloaded*	Average speed
User #1	1.706,36 ms	22	21	0	252.128.709	56.288,71 kbit/s
User #2	1.688,96 ms	20	19	0	228.116.451	56.868,81 kbit/s
User #3	1.855,41 ms	17	16	0	192.098.064	51.767,15 kbit/s
User #4	2.723,30 ms	14	13	0	156.079.677	35.269,36 kbit/s
User #5	2.703,80 ms	12	11	0	132.067.419	35.523,66 kbit/s
User #6	3.150,05 ms	10	9	0	108.055.161	30.491,26 kbit/s
User #7	2.215,72 ms	8	8	0	96.049.032	43.348,85 kbit/s
User #8	2.991,57 ms	6	5	0	60.030.645	32.106,58 kbit/s

Table 5.1: Download data per user.

*Size in Bytes

Tabel 5.1 describes metrics for each user. As the test was a ramp up, the first users had more active time, which results in more requests. As the these first users made requests while the supernode was (mostly) iddle, their average download time is quite low and their average speed is high when compared to the last users to become active. The average time and speed is similar for the first 3 users, while the other 5 users show a much higher average time and lower download speed. Of the latter group of users, User #7 has better metrics due to being the only one that finished all the requests, while all the other 7 users had a download ongoing when the test stopped.

While this test was performed the CPU usage of the supernode remained at 2%. The upload output was at 11,35MB/s, a little shy of the 11,92MB/s maximum network limit. High HTTP outputs of data have almost no stress on the supernode CPU-wise. As for bandwidth output, this test concludes that bandwidth usage is not a limiting factor for the supernode as long as it has bandwidth available.

Delay on DHT Requests

In this test there is data on the delay of DHT requests that are made by supernodes for the BitTorrent functionality. The client supports trackers, but for this test no trackers were used and the supernode would rely entirely on the DHT to join the peer swarm. This means that after a torrent is loaded in the SN, connecting to peers happens only after a DHT request is finished successfully.

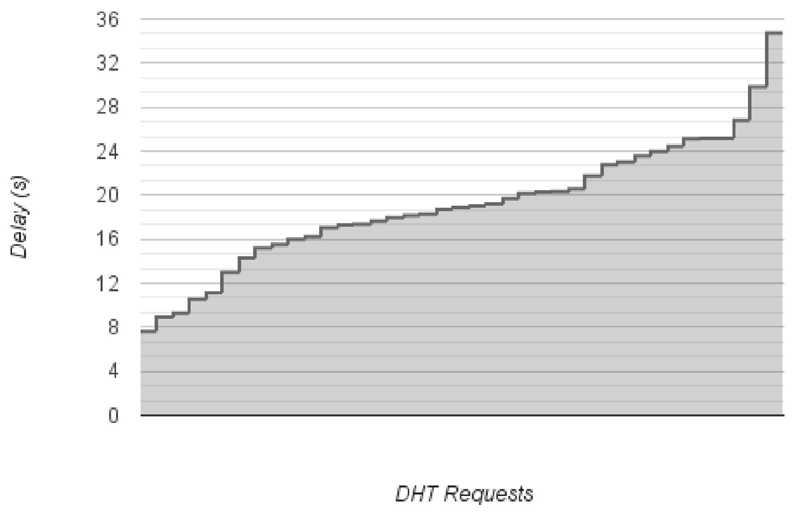


Figure 5.38: Delay of DHT Requests.

This test used the current DHT implementation from the SN's BitTorrent client, the objective of this test was to have an idea of the order of magnitude of the delay that DHT implementations take until the request is made until a full swarm list of peers is retrieved. The average of this test was in the 18 second window of delay between the request being made and the list being retrieved, being that the highest delay was 34 seconds. Given the decentralized nature of this solution an average 18 second window seems a high delay but it is an acceptable delay for file sharing solutions given that no centralized server is used and that the DHT Network is composed of 15 to 27 million concurrent nodes. [53]

A simple Kademlia DHT implementation was also tested in early stages, this DHT implementation was supposed to be used on the SN Network for the search functionality between supernodes. The average request under this implementation yielded worse results, usually above 20 seconds, data from this implementation wasn't furtherly gathered as the functionality isn't used on the final prototype.

Chapter 6

Conclusions

It was a challenging project due to the nature of the protocols involved and their lack of being mature as a technology, this led to a lot of headaches and hacky solutions that due to updates on the browsers and WebRTC API, left a lot of the code with room for structural changes. The P2P nature of the dissertation was very fun to work with, examples like clients under diverse and distinct networks behaved inconsistently despite WebRTC typically working behind NATs and firewalled networks, or how peers sometimes behaved differently in a P2P environment even with the same code. These as well as working with an API that was ever changing proved to be challenging but very educational.

The goal of this prototype at the time of its birth, from idea to its completion was to give a new perspective to the BitTorrent, a way to turn file sharing more client-free and an uncensored access to content, specially after all the backlash in this recent years to BitTorrent and P2P in general. While Seedseer was being developed a multitude of similar solutions started to show up, which shows a real need for these types of projects and validates the approach that this dissertation followed on the subject of different and simpler ways to use and consume content in P2P networks. The solution however begs new questions, being that the seedseer network is more fragmented and harder to scan is a plus, but when a peer is connected to another, the use of WebRTC raises concerns over privacy and lack of anonymity, would users consider this aspect to be a valuable change when compared to BitTorrent? The ease of use over a browser allows, when compared to other P2P solutions, for a potential higher number of clients but with more limited resources and a higher rate of peers joining and leaving the network, to what degree would this be considered an advantage or a disadvantage? The answer to these rely on user behavior and to what extent the user values these changes as advantages over other existing P2P solutions. This dissertation aimed for a different, more accessible P2P solution over a browser, while knowingly sacrificing performance, this is exactly what it achieved.

6.1 Future Work

There are a lot of concepts that were unpolished on the development of Seedseer, if this prototype was to hit production and be used in a more serious environment a lot of obvious enhancements would be required. The improvements can be divided in 2 groups, improvements that are related to how BitTorrent works and improvements into Seedseer's core concepts. The first are features that Seedseer can use that BitTorrent clients already use, features that are stable and have already been proved to work successfully in the large network of nodes that is BitTorrent as a protocol. The second are changes into the Seedseer core or architecture to mitigate its inherent limitations.

Regarding the first group, Seedseer would achieve a greater maturity if its WebRTC transfers were brought closer to the BitTorrent way of handling file transfers. Features like data integrity verification per piece, optimization of piece requests, superseed and endgame mode, all of these allow a better use of the network's resources, resilience to ill intentioned peers and assure better quality of service for regular peers in the network, using concepts that BitTorrent has that have been proved to be successful. As for the second group, a swifter approximation between the BitTorrent and WebRTC network would be an improvement to make the solution more resource effective, meaning a less need for overhead or structure changes between the 2 protocols, making so file sharing between different networks to be as seamless as possible. The SuperNode Network would be a core step for communication between supernodes using websockets, being the two major features, allowing content search through a DHT and routing WebRTC signaling between peers connected to different supernodes. However, limiting its range of features and maximum number of hops would be essential to determine how well each seedseer network would scale and its size. Finally, as seedseer's concept relies on being available in a browser and clients should be more limited on resources than the typical BitTorrent client, this means RAM Optimization and disk IO usage is essential for its effective adoption by the users, this change alone determines heavily how much content can be available on the network and be seeded effectively.

Bibliography

[1] Scrape - vuzewiki. https://wiki.vuze.com/w/Scrape, 2012. (accessed August 24, 2016).

[2] The world in 2015: Ict facts and figures. https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf, 2015. (accessed August 2, 2016).

[3] Bittorrent protocol specification v1.0. https://wiki.theory.org/BitTorrentSpecification, 2016. (accessed July 29, 2016).

[4] Internet live stats. http://www.internetlivestats.com/internet-users/, 2016. (accessed July 29, 2016).

[5] Acorn, J. Forensics of bittorrent.

[6] Adamsky, F., Khayam, S. A., Jäger, R., and Rajarajan, M. P2p file-sharing in hell: Exploiting bittorrent vulnerabilities to launch distributed reflective dos attacks. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (Washington, D.C., Aug. 2015), USENIX Association.

[7] Aitken, D., Bligh, J., Callanan, O., Corcoran, D., and Tobin, J. Peer-to-peer technologies and protocols. http://ntrg.cs.tcd.ie/undergrad/4ba2.02/p2p/, 2001. (accessed July 29, 2016).

[8] Akoka, J. Centralization versus decentralization of information systems : a critical survey.

[9] Alberty, T. The new ponzi scheme: Bittorrent & hardcore pornography.

[10] Alvestrand, H. Google release of webrtc source code. https://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html, 2011. (accessed July 29, 2016).

[11] Bartholomew, T. B. The death of fair use in cyberspace: Youtube and the problem with content id. 66–88.

[12] Bieber, J., Kenney, M., Torre, N., and Cox, L. P. An empirical study of seeders in bittorrent. 67–70.

[13] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *INTERNATIONAL WORKSHOP ON DESIGNING PRIVACY ENHANCING TECHNOLOGIES: DESIGN ISSUES IN ANONYMITY AND UNOBSERVABILITY* (2001), Springer-Verlag New York, Inc., pp. 46–66.

[14] Cohen, B. Incentives build robustness in bittorrent, 2003.

[15] da Silva Rodrigues, C. K. Analyzing peer selection policies for BitTorrent multimedia on-demand streaming systems in internet. *IJCNC 6*, 1 (jan 2014), 203–221.

[16] Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M. F., and Morris, R. Designing a dht for low latency and high throughput. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1* (Berkeley, CA, USA, 2004), NSDI'04, USENIX Association, pp. 7–7.

[17] danah m. boyd, and Ellison, N. B. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication 13*, 1 (oct 2007), 210–230.

[18] Decker, C., Eidenbenz, R., and Wattenhofer, R. Exploring and improving bittorrent topologies. 13-th IEEE International Conference on Peer-to-Peer Computing.

[19] Feldman, M., Lai, K., Stoica, I., and Chuang, J. Robust incentive techniques for peer-to-peer networks. In *Proceedings of the 5th ACM Conference on Electronic Commerce* (New York, NY, USA, 2004), EC '04, ACM, pp. 102–111.

[20] Fette, I., and Melnikov, A. Rfc 6455 - the websocket protocol. `https://tools.ietf.org/html/rfc6455`, 2011. (accessed July 29, 2016).

[21] Harrison, D. Peer id conventions. `http://www.bittorrent.org/beps/bep_0020.html`, 2008. (accessed July 29, 2016).

[22] Harrison, D. Index of bittorrent enhancement proposals. `http://www.bittorrent.org/beps/bep_0000.html`, 2012. (accessed August 22, 2016).

[23] Herrmann, M., Zhang, R., Ning, K.-C., Diaz, C., and Preneel, B. Censorship-resistant and privacy-preserving distributed web search. In *14-th IEEE International Conference on Peer-to-Peer Computing* (sep 2014), Institute of Electrical & Electronics Engineers (IEEE).

[24] Hickson, I., and Hyatt, D. Html 5: A vocabulary and associated apis for html and xhtml. `https://www.w3.org/TR/2008/WD-html5-20080610/single-page/`, 2008. (accessed July 29, 2016).

[25] Hugoson, M.-Å. Centralized versus decentralized information systems. In *IFIP Advances in Information and Communication Technology*. Springer Science Business Media, 2009, pp. 106–115.

[26] Johnsen, J. A., Karlsen, L. E., and Birkeland, S. S. Peer-to-peer networking with bittorrent. 1–20.

[27] Khan, M. Stun or turn? which one to prefer; and why? `https://www.webrtc-experiment.com/docs/STUN-or-TURN.html`, 2013. (accessed July 29, 2016).

[28] Leitão, J., Carvalho, N. A., Pereira, J., Oliveira, R., and Rodrigues, L. On adding structure to unstructured overlay networks. In *Handbook of Peer-to-Peer Networking*. Springer Science Business Media, oct 2009, pp. 327–365.

[29] Levent-Levi, T. What is webrtc? `https://bloggeek.me/webrtc/`, 2012. (accessed July 29, 2016).

[30] Liang, J., Kumar, R., and Ross, K. W. Understanding kazaa, 2004.

[31] Liberatore, M., Erdely, R., Kerle, T., Levine, B. N., and Shields, C. Forensic investigation of peer-to-peer file sharing networks. *Digital Investigation 7* (aug 2010), S95–S103.

[32] Meulpolder, M., D'Acunto, L., Capotă, M., Wojciechowski, M., Pouwelse, J. A., Epema, D. H. J., and Sips, H. J. Public and private bittorrent communities: A measurement study. In *Proceedings of the 9th International Conference on Peer-to-peer Systems* (Berkeley, CA, USA, 2010), IPTPS'10, USENIX Association, pp. 10–10.

[33] Neglia, G., Presti, G. L., Zhang, H., and Towsley, D. F. A network formation game approach to study bittorrent tit-for-tat. In *Network Control and Optimization, First EuroFGI International Conference, 2007, Avignon, France, June 5-7, 2007, Proceedings* (2007), pp. 13–22.

[34] Office of the United Nations High Commissioner for Human Rights. The promotion, protection and enjoyment of human rights on the internet, 2014.

[35] Office of the United Nations High Commissioner for Human Rights. The right to privacy in the digital age, 2014.

[36] Perrin, A. Social media usage: 2005-2015. `http://www.pewinternet.org/files/2015/10/PI_2015-10-08_Social-Networking-Usage-2005-2015_FINAL.pdf`, 2015. (accessed August 2, 2016).

[37] Piatek, M., Isdal, T., Anderson, T., Krishnamurthy, A., and Venkataramani, A. Do incentives build robustness in bit torrent. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association, pp. 1–1.

[38] Pick, R. A. Shepherd or servant: Centralization and decentralization in information technology governance. *IJMIS 19*, 2 (mar 2015), 61.

[39] Portmann, M., and Seneviratne, A. Cost-effective broadcast for fully decentralized peer-to-peer networks. *Computer Communications 26*, 11 (jul 2003), 1159–1167.

[40] Postigo, H. Capturing fair use for the youtube generation: The digital rights movement, the electronic frontier foundation and the user-centered framing of fair use. *Information, Communication & Society 11*, 7 (2008), 1008–1027.

[41] Pouwelse, J., Garbacki, P., Epema, D., and Sips, H. The bittorrent P2P file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*. Springer Science Business Media, 2005, pp. 205–216.

[42] Qiu, D., and Srikant, R. Modeling and performance analysis of bittorrent-like peer-to-peer networks. *SIGCOMM Comput. Commun. Rev. 34*, 4 (Aug. 2004), 367–378.

[43] Saint-Andre, P. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, Oct. 2015.

[44] Sandvine. Global internet phenomena report africa, middle east & north america. `https://www.sandvine.com/downloads/general/global-internet-phenomena/2015/global-internet-phenomena-africa-middle-east-and-north-america.pdf`, 2015. (accessed August 24, 2016).

[45] Sandvine. Global internet phenomena report asia-pacific & europe. `https://www.sandvine.com/downloads/general/global-internet-phenomena/2015/global-internet-phenomena-report-apac-and-europe.pdf`, 2015. (accessed August 24, 2016).

[46] Sharma, A. K., and Sharma, A. P. N. Bittorrent (peer topeer network): Antipiracy and anonymity.

[47] Shen, H., Liu, A. X., and Zhao, L. Freeweb: P2p-assisted collaborative censorship-resistant web browsing. *2014 43rd International Conference on Parallel Processing 0* (2013), 130–139.

[48] Steinmetz, R. *Peer-to-peer systems and applications.* Springer, Berlin New York, 2005.

[49] Tadlock, C. Copyright misuse, fair use, and abuse: How sports and media companies are overreaching their copyright protection.

[50] Tucker, D. Survey of searching methods in internet peer-to-peer systems. `http://medianet.kent.edu/surveys/IAD03F-dtucker/index.html`, 2003. (accessed July 29, 2016).

[51] van der Spek, O., and Norberg, A. Bittorrent udp-tracker protocol extension. `http://www.libtorrent.org/udp_tracker_protocol.html`, 2016. (accessed July 29, 2016).

[52] Wang, J., Shen, R., Ullrich, C., Luo, H., and Niu, C. Resisting free-riding behavior in bittorrent. *Future Gener. Comput. Syst. 26*, 8 (Oct. 2010), 1285–1299.

[53] Wang, L., and Kangasharju, J. Measuring large-scale distributed systems: case of BitTorrent mainline DHT. In *IEEE P2P 2013 Proceedings* (sep 2013), Institute of Electrical & Electronics Engineers (IEEE).

[54] Winter, A. Downloaded, 2013.

[55] Wolchok, S., and Halderman, J. A. Crawling bittorrent dhts for fun and profit. In *Proceedings of the 4th USENIX Conference on Offensive Technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–8.

[56] Zhang, H., Ye, L., Shi, J., Du, X., and Chen, H. H. Preventing piracy content propagation in peer-to-peer networks. *IEEE Journal on Selected Areas in Communications 31*, 9 (September 2013), 105–114.