Distributed Systems
& Ledgers Lab

Peer-to-Peer Content Delivery Network

Student Research Project

Project Team:	Adrian Locher
	Jason Benz
Project Advisor:	Dr. Thomas Bocek
Proofreader:	AnneMarie O'Neill
Date:	22.12.2022

OST
Eastern Switzerland
University of Applied Sciences

Abstract

Web applications have historically been centralized in their way of distributing data. Peer-to-peer protocols such as BitTorrent have only recently been introduced to the web thanks to the increasing support of WebRTC. This project analyses technologies that already take advantage of this while proposing improvements to increase decentralization.

After researching available technologies and proposing a new concept, we built our proposed concept as a prototype. The focus of this prototype is to be as decentralized as possible, while still working with a web application without the need to install any plugins or external software by a user.

Our concept, backed by the prototype, shows that it is possible to implement a system for delivering files in a peer-to-peer fashion without centralized services. Arguably, the developed prototype is not suitable for small, latency sensitive data, because of the latency introduced by the complex nature of peer-to-peer connection establishment. With current technologies, a sub-second download of any file is therefore impossible.

Latency can still be improved by prioritizing peers for latency, to reach smaller round-trip times and therefore faster connection-establishment. This mechanism remains to be solved by future work.

Lay Summary

Large companies often provide files over so-called content delivery networks. They are hosted and managed centrally by the corresponding company. The files are often stored in different locations. This has disadvantages. Several servers have to be operated for this purpose and a single company decides which content is available.

This thesis addresses whether a completely decentralized web-based peer-to-peer CDN can be implemented. The goal is that an internet user automatically joins the CDN as soon as they open the corresponding website. The user should not have to install any programs to participate.

In the first part of the work, it is researched if it is technically possible. In the second part, the research is proven with a prototype. Modern web technologies are used for the implementation.

Acknowledgements

During the student research project, Dr. Thomas Bocek and AnneMarie O'Neill supported us in their areas of expertise, and we would like to express our sincere thanks to them.

Many thanks to Dr. Thomas Bocek for mentoring us and providing input especially in the domain of distributed systems.

We very much appreciate AnneMarie O'Neill proofreading the document and providing valuable feedback.

Glossary

Name	**Abbr.**	**Description**
Content Delivery Network	CDN	Group of servers which are geographically distributed to efficiently distribute content.
Distributed Hash Table	DHT	Distributed data structure that supports a data lookup based on a key-value pair.
Interactive Connection Establishment	ICE	A combination of the STUN and TURN protocols.
Leecher	-	A peer which is downloading content from a P2P network (coll. taker).
Network Address Translation	NAT	A technology used by routers to reduce the amount of needed public IP-addresses.
Node	-	Synonym of peer.
Open Systems Interconnection model	OSI model	Layer model used by systems to communicate over a network.
Peer	-	Endpoint for communication in a distributed computer system. Simultaneously has the role of a client and a server. Peers are the total amount of seeders and leechers.
Peer-to-Peer	P2P	Distributed application architecture in which participants (peers) communicate ”face-to-face”.
Piece	-	Part of a data-unit to share via P2P protocols.
Round Trip Time	RTT	The time needed from when a request is sent until a response is received.
Seeder	-	A peer which is uploading content to a P2P network (coll. giver)
Session Traversal Utilities for NAT	STUN	A service for a node to discover its own IP-address and port while behind NAT.
Swarm	-	A set of peers seeding the same files.
Torrent	-	A data-unit, seeded via the BitTorrent protocol described by a meta-info file (.torrent file).
Traversal using Relays around NAT	TURN	A relays service to connect to peers while behind symmetric NAT.

Table 1.: Glossary.

Contents

Part I.

Paper

1. Introduction

Nowadays, companies often use a CDN to quickly make large content available to users. CDNs are servers which are geographically distributed. Many companies use services such as Cloudflare or operate their own CDN.

The servers are distributed, but not decentralized. Content is provided multiple times in different CDNs, which are all centrally managed. A prominent example is the distribution of jQuery or Google Fonts. Often, large content such as images or videos are also distributed this way.

Content distribution could be solved by using a P2P network. Frequently used content is most likely already available from a nearby peer. The problem when using a P2P network is that the clients either have to install a torrent client first or have severe limitations in web-based solutions. The goal of this work is to research existing solutions and propose improvements.

The work will be divided into the following three parts:

1. Determining existing technologies.
2. Creating a concept with a proposal of a solution.
3. Creating a prototype to identify the technical limitations and to prove the concept.

2. Background Information

There are a lot of different approaches to establish a P2P connection to share files. It is possible with different technologies and protocols. Technologies such as BitTorrent are often associated with illegality. The media reports about legal warnings and penalties because someone has downloaded files over P2P. Especially in the USA and in Germany this is the case. The use of P2P file-sharing in general is not illegal. However, it is illegal to share certain kinds of content such as malware or copyright infringing files.
Legally, it is possible to implement a CDN via P2P. What remains is the question regarding practicability. This question will be explored in depth. This document will determine which technologies exist, which are suitable and what technical limitations are encountered.

2.1. BitTorrent

BitTorrent is a P2P file sharing protocol [1]. The main goal of this protocol is to download large files in a distributed manner. While a classic download approach using the Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP) only downloads data from one source after another, BitTorrent establishes a decentralized network called a swarm for each file. Individual parts of the desired file are downloaded in parallel from different peers at the same time, which might even increase bandwidth. At the end, the individual file parts are reassembled. Unless trying to leech, the parts being downloaded are simultaneously made available (seeded) to other peers so that they can also benefit from the network.

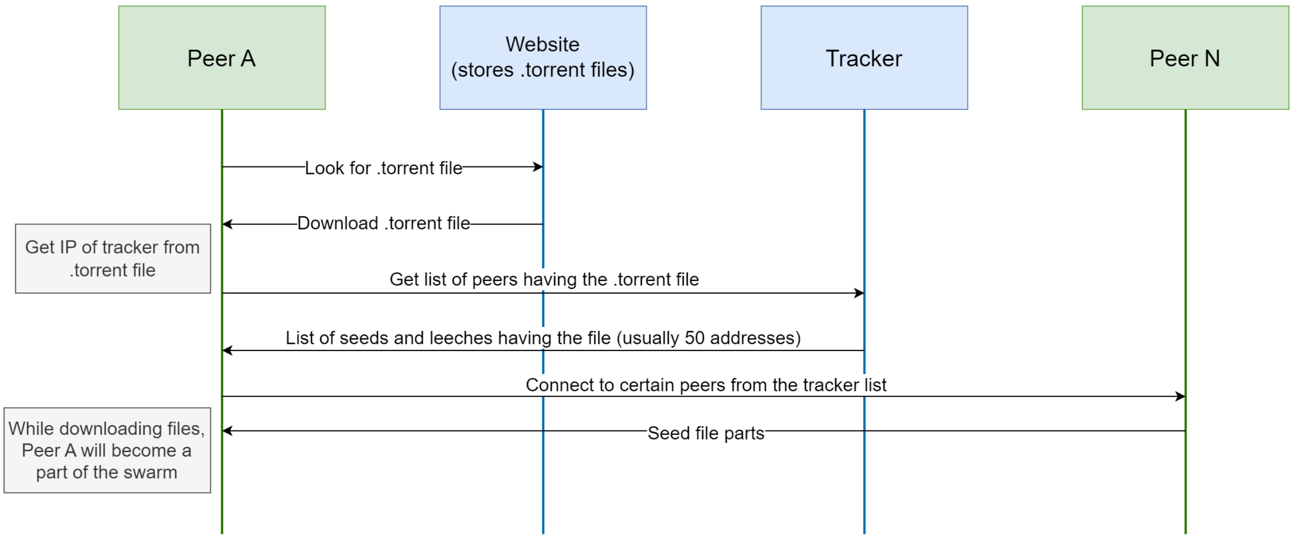

Peer A
Website (stores .torrent files)
Tracker
Peer N
Look for .torrent file
Download .torrent file
Get IP of tracker from .torrent file
Get list of peers having the .torrent file
List of seeds and leeches having the file (usually 50 addresses)
Connect to certain peers from the tracker list
Seed file parts
While downloading files, Peer A will become a part of the swarm

Figure 2.1.: BitTorrent Connection Establishment.

Advantages	**Disadvantages**
+ Performance (more bandwidth)	- NAT problems
+ Decentralized	- Possible malware sharing / copyright violation
+ Easily scalable due to swarming	- In most cases, a client is required to connect

Table 2.1.: BitTorrent Advantages and Disadvantages.

According to BitTorrent estimates, over 250 million users utilize this technology every month. This corresponds to about 3.35% of internet traffic [2].

2.2. WebRTC

Web Real-Time Communication (WebRTC) is a collection of protocols and APIs that are combined into an open standard. It enables real-time communication between browsers from other clients (P2P). To establish a link between the peers, a connection to a signaling server must first be established. This connection can be made through a conventional way, such as WebSockets. However, any medium can be used for signaling. For instance, audio transmission could also be used [3].

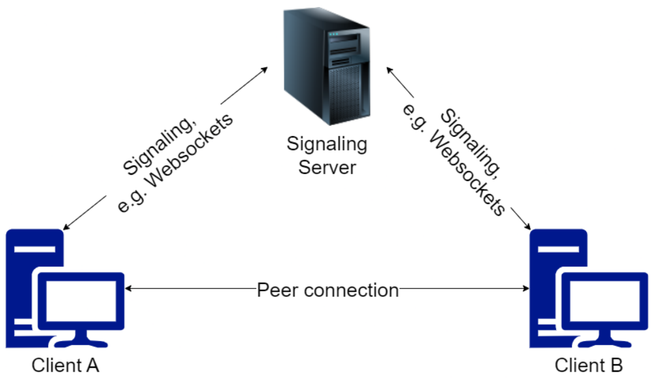


Figure 2.2.: Basic WebRTC Architecture.

Advantages	**Disadvantages**
+ P2P connection in web browsers	- Caching is only possible with restrictions
+ Open web standard	- Signaling server is required to establish a connection
+ WebRTC is integrated in all major browsers and enabled by default	- Security (possibility to get the clients IP addresses)
+ There is no need to download a specific client	

Table 2.2.: WebRTC Advantages and Disadvantages.

2.3. WebTorrent

WebTorrent is the largest web based torrent client. It is written in JavaScript and works in the browser. The source code is available on GitHub (MIT license) [4]. The GitHub project includes the full client and the repositories of the different used libraries such as a DHT implementation, a BitTorrent peer wire protocol implementation, a tracker implementation, a desktop client, etc. WebTorrent is based on a modified version of the BitTorrent protocol. Therefore, it behaves mostly like BitTorrent, with the main difference that it uses WebRTC as a transport protocol. Hybrid WebTorrent clients can also interact with BitTorrent clients.

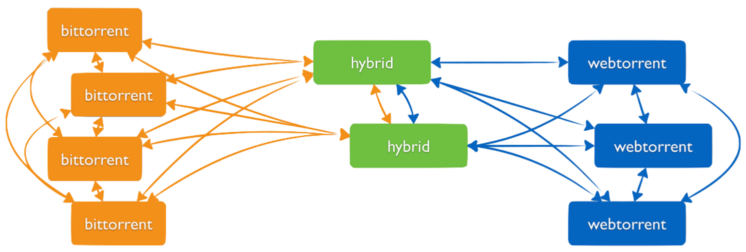


Figure 2.3.: WebTorrent Peer Interaction Map [5].

Advantages	**Disadvantages**
+ P2P connection in web browsers	- Details are not well documented
+ Comprehensive implementation is available	- Only limited configurability (= adjustments are time-consuming)
+ Active open source project	
+ Hybrid solutions possible (BitTorrent / WebTorrent)	

Table 2.3.: WebTorrent Advantages and Disadvantages.

2.4. Hypercore

The Hypercore Protocol is another P2P approach to file-sharing [6]. It combines a lightweight blockchain without a consensus algorithm crossed with a BitTorrent like protocol. It consists of the following subprojects:

- Hypercore: A distributed append-only log, which is like a list where only data can be added. It is secured by a hash tree using cryptography
- Hyperdrive: A P2P file system built up on Hypercores. Usually the users only download the file parts needed
- Hyperswarm: A DHT implementation based on Kademlia (including hole punching)

The project was inspired by BitTorrent.

Advantages	**Disadvantages**
+ Active open source project	- Not so widely used
+ Very well documented	- Limited application (compatibility)

Table 2.4.: Hypercore Advantages and Disadvantages.

2.5. IPFS

Another interesting project is the InterPlanetary File System (IPFS). The self-proclaimed mission of the project is:

> *A peer-to-peer hypermedia protocol designed to preserve and grow humanity's knowledge by making the web upgradeable, resilient, and more open. [7]*

While the ambitions of this project are very high, in its core it is a distributed, P2P file system. IPFS is structured similarly to a single large BitTorrent swarm, which exchanges objects via a large repository-like system. The hash of the files acts as a content-ID (CID). The CID changes when the file content changes. To keep focus on the files, the InterPlanetary Naming System (IPNS) was developed. IPNS employs asymmetric cryptography to create mutable pointers to CIDs, enabling an addressing-scheme that allows for files to change while the address stays the same. To be able to interact with IPFS, downloading a client is required.

Advantages	**Disadvantages**
+ Full implementation available	- Not yet mainstream
+ Active open source project	- Client download required
+ IPFS nodes can be owned by a company	

Table 2.5.: IPFS Advantages and Disadvantages.

2.6. PeerCDN

Feross Aboukhadijeh, the developer of WebTorrent, has already developed a P2P CDN. However, this was sold to Yahoo many years ago. The goal was to make the project well-known. Nowadays, no more information can be found about it. Either the project was transferred to closed source or terminated completely. [8]

3. Concept

This chapter describes the concept of a P2P-CDN, later to be implemented as a prototype. The concept builds upon existing technologies and proposes improvements upon them.

3.1. Requirements and Decisions

The main goal of this thesis is to develop a concept for a P2P CDN. The team has set itself a number of requirements that have a significant impact on the decisions it makes:

- No expertise is required for using the CDN ("Grandma should be able to use it").
- The system is scalable.
- The system is decentralized.
- Modern technologies are in use.
- The latency is as low as possible.
- No client needs to be downloaded.

These requirements limit the determined technologies in Chapter 2. PeerCDN can be excluded directly as no information is accessible. An existing BitTorrent client, IPFS and Hypercore can be excluded as well, as a client must be downloaded. In addition, the technologies mentioned require a certain amount of knowledge in order to work well with them. WebTorrent uses proven technologies with BitTorrent and modern ones with WebRTC and a browser-based approach. There is also no need to install a client. At first glance, it looks like the perfect solution. However, a closer analysis reveals that there are some limitations. WebTorrent has very limited configurability. There are also limitations in the use and optimization of DHTs as a peer-discovery mechanism. Nevertheless, this technology is very suitable as a basis. This chapter attempts to describe how to connect the individual open source libraries with additional libraries and self-developed components. Thus, an adaptation for the specific use case of a fully decentralized P2P CDN can be developed. In the following sections a concept is proposed on how to build a P2P CDN by employing modern technologies and libraries while pointing out their drawbacks and benefits.

3.2. Components and Considerations

In this chapter, the document explains which components are needed to build a fully decentralized, in-browser P2P-CDN. For each component the considerations that have to be undertaken are summarized.

3.2.1. Peer Protocol

The BitTorrent Peer Protocol can be implemented transport agnostic, as is done by the developer behind WebTorrent [9]. For our purpose the protocol will be tunneled through a WebRTC connection. This approach is similar to WebTorrents browser-to-browser connections.

At this time, true sockets cannot be used by web applications natively inside any of the major browsers. There is a proposal in the Web Incubator Community Group [10], created in 2020. Currently, none of the major web-browsers seem to be implementing a feature like this. The main reason for this are security concerns as listed in the discussion about a potential "Raw Sockets API" [11] of the Mozilla standards-positions. The request was eventually marked as "Harmful" by Mozilla. The main concern seems to be that this would allow a web application to circumvent security mechanisms like the *Same Origin Policy*.

Although, as elaborated, there is no support for "raw" sockets natively, Firefox and Google Chrome allow for message passing to system-local applications to be used via extensions [12][13]. An approach to employ this for network-communication was undertaken by a project called Socketify [14]. One major disadvantage of this approach is that there needs to be a client installed on the system using the extended browser.

The only true native P2P mode of transportations remains WebRTC, which is why further in this text, it is assumed that WebRTC is used to achieve the requirements to the system described in this document.

A possibility to improve the performance of data-transport is to prioritize peers with low RTT and high bandwidth. While both RTT and bandwidth can simply be measured during data-transport. It might be beneficial to pre-prioritize the peer list, before even starting to communicate with a peer. This can be achieved by introducing a concept of geographical closeness. Approximate two-dimensional coordinates could be encoded into the peer ID, by mapping them to one-dimensional space, for example by employing Hilbert's space-filling curve [15].

3.2.2. Peer Discovery

BitTorrent applies three ways to receive information about peers of a certain swarm.

Trackers [16] are servers that store and serve peer information. Since this is a client-server architecture, implementing a tracker-client for the web is easy. However, this does not meet our goals on its own, since trackers are centralized instances and, apart from reliability concerns, could be easily censored.

PEX or Peer-Exchange [17] is a functionality of sharing information between the downloading peers themselves. While being one of the most effective ways of distributing peer-information, this still relies on an information source outside the swarm for bootstrapping purposes.

The DHT Protocol [18] is an implementation of the Kademlia DHT [19]. The DHT is used to store pairs of [info-hash; peer-information] and can be applied as a replacement for trackers.

Since WebRTC based applications cannot establish connections to different in-browser applications without having a signaling service, it is impossible to integrate DHT based on in-browser clients only. Even directly querying an external DHT from within browsers is inefficient, since the overhead of currently enabled transports is too high to route through a DHT with sufficient latency for most tasks. This is why even sophisticated projects like WebTorrent have not yet implemented it as pointed out in issue #288 of the WebTorrent repository [20] and by the Hyper Hyper Space project [21].

As an in-browser application, WebTorrent uses trackers as a primary peer-information distribution mechanism. The tracker server also acts as a signaling server, limiting peer connections between peers that use at least one common tracker, and between peers that find each other via PEX.

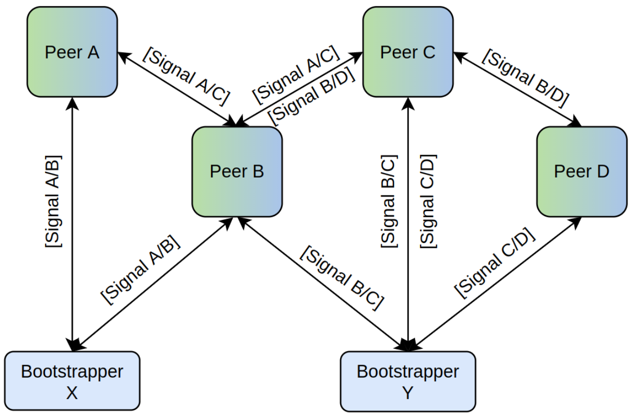


Figure 3.1.: Overview about how an In-Browser DHT Would Work.

As seen in Figure 3.1, an in-browser DHT implementation would still need to rely on external bootstrapping nodes that could be reached via HTTP or WebSocket. Furthermore, assuming the DHT would follow a mechanism based on Kademlia, the network would have to endure the overhead of signaling for each hop during routing.

3.2.3. Signaling

P2P-applications between browsers are mostly built upon WebRTC [22]. While WebRTC does provide a means to communicate P2P between two clients running in a browser environment, most applications still use a centralized signaling service to initiate the connection.

There have been successful attempts to implement signaling mechanisms via DHTs. For instance, the previous work done at OST [23] proposes an architecture whereby web services provide an additional DHT service external to the in-browser application.

This seems like a sufficient solution for our case, since there is still enough decentralization. While the network for the signaling mechanism will not have as many nodes as there are clients, it will be possible to have the same amount of nodes and web services. Our architecture does not dismiss the idea of web services altogether, but it focuses on content delivery of big files. Therefore, the decentralization of the potential system is not limited stronger than the amount of web services taking part in it, as long as every participating web service hosts a node in a DHT based signaling mechanism as well.

3.2.4. NAT Traversal

To traverse NAT (Network Address Translation), WebRTC uses ICE (Interactive Connection Establishment), which consists of STUN (Session Traversal Utilities for NAT) and TURN (Traversal using Relays around NAT).

Asymmetric NAT allows for a node different from the requested node to answer a request. In order to establish a connection the internal node needs to first send a request to a known node. This STUN-node also tells the internal node its public IP-address and port which it needs to communicate to the signal server how it may be reached.

Symmetric NAT only allows external nodes to traverse if they (a node being identified by an IP-address and port) were the target of a request by an internal node. In this case, connections have to be established via a relay that forwards traffic to the respective connection partners additional to using STUN. This mechanism is called TURN.

ICE Candidates are inherently centralized but can be configured by the node offering signaling information. To provide maximum reliability, it would make sense to deploy ICE-services for every web service deployed.

3.2.5. Mediation

We propose an additional service to web services called mediation, which combines signaling- and peer-discovery. This service is located outside web browsers and has to employ raw-sockets as transport. Mediation is split into two sublayers: The service layer and the replication layer. The service layer acts as a tracker, hosting peer information in the form of [info-hash ; signaling-data]. The replication layer takes part in a DHT, to share tracker- and signaling information between mediators. This combines the speed of using a tracker, with the reliability of using a distributed system. Web services with small amounts of concurrent visitors, for instance, can still profit from the network, if different websites (ideally with more concurrent users) deploy the same files.

3.2.6. DHT

BitTorrent, as well as most P2P file sharing applications, use some form of Kademlia DHTs [19]. While Kademlias proven consistency and efficient lookup algorithms are beneficial, its distance metric leads to unpredictable latency. Kademlia uses the XOR operation as its measure of distance between a randomly generated node ID and the DHT-Key to be found in lookups. This effectively means that two *close* Kademlia nodes may be on opposite sides of the planet. This causes problems for this project, whenever peers of latency sensitive files can only be found via the DHT mechanism.

One improvement on this topic was proposed in a paper by Raul Jimenez et al. [24]. The paper discovered significant latency improvements by adapting different concepts of the original specification.

The biggest improvements were seen after applying a concept named NR128, where bucket sizes are changed from a standard size k for all buckets to $k1 = 128$ for the closest, $k2 = 64$ for the second closest to the own ID, etc. Until the rest of the buckets remain at $k = 8$.

Another way to address latency in DHTs would be to approximate it by the IDs used by nodes that may be contacted. One approach to this was described by Ratti et. al. in their paper about NL-DHT [25]. Their concept introduces latency aware node IDs by employing a mapping from three-dimensional location space to the one-dimensional space that is used by nodes as their ID. This makes it possible to approximate the latency to nodes and ultimately allowed them to reduce the hop-count for lookup operations.

While the approach used in NL-DHT [25] certainly reduced the latency by some amount this also seems to introduce a new problem. In standard Kademlia as described in the original paper [19], nodes may choose their ID by a uniform distribution function. This means that if a key is stored on k nodes, some nodes might be geographically far away while others may be very close. Using location aware IDs on the other hand leads to all nodes storing a key to be geographically close to each other. Any node contacting the nodes storing the key will receive answers with approximately the same latency from each node.

Adding predictable latency to the system is still an important step towards less latency. Based on the suggestions made in the NL-DHT paper [25], we propose a mechanism that might further improve latency by adding a feature of distribution of keys to the concept. As previously discussed, location awareness can be reached by mapping the n dimensions of geographical space to single-dimensional ID-space. We propose adding a dimension for key-space to those n dimensions to still have the advantages of randomly distributed keys. This n+1 dimensional space would cover all geographical dimensions plus one uniformly distributed dimension that has the same size as the key-space, assuring that keys are distributed uniformly across the world, while still ensuring deterministic latency. Formally, this can be defined as separating the concept of distance between nodes and keys from distance between nodes and other nodes.

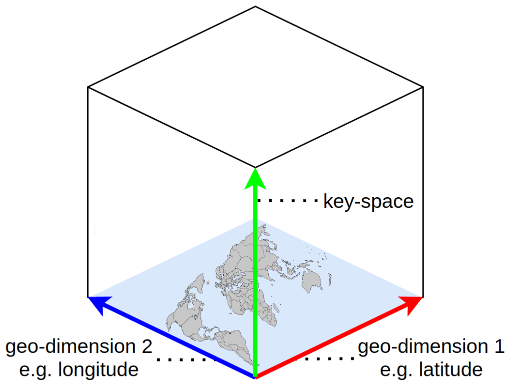


Figure 3.2.: Space of a DHT-Node ID Split into its Dimensions.

3.2.7. Caching

Files downloaded inside the browser via a system like the one described in this document, are not cached natively by a browser since those files are not downloaded from a single server anymore. Still, caching could highly improve the performance of the system, not only because the peer that is caching the file could simply reuse it when needed, but also because mentioned peer could again seed the files to other peers that need them.

Furthermore, since our P2P-CDN system assumes that different web applications might want to deploy the same file, it would be helpful if the same browser instance could cache a file for all web applications using the system making it possible to reuse a file downloaded after instruction by web application X, again while being instructed to download the same file by web application Y.

The most sophisticated way at this time is the IndexedDB API [26], provided by most modern web-browsers. This API provides a means to store large files or "blobs" on the client-side of a web application. Unfortunately IndexedDB databases do not allow for cross domain access for security reasons. This means that files can only be cached per web application instance.

Smaller files could also be cached by the "localStorage"-API which is usually limited to 10 MB of storage, but more light-weight and easier to use. In general, the IndexedDB API still seems like the most reasonable choice for potentially big files.

3.3. Architecture

The following is a conceptual architecture for a fully-decentralized in-browser P2P-CDN.

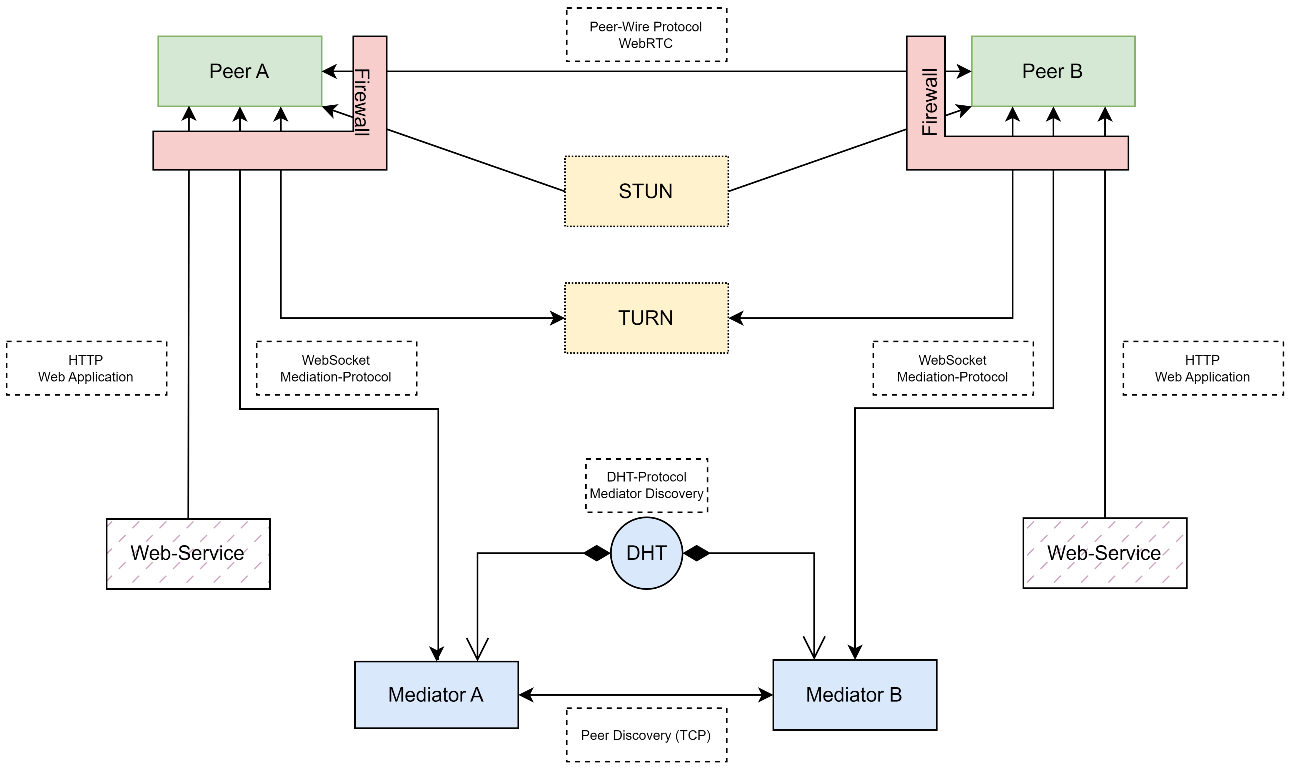


Figure 3.3.: Architecture Overview of the Concept.

3.3.1. Mediation Protocol

The mediation protocol is used between a peer and its mediator, as well as between mediators to exchange information about peers and how to contact them and to transport signaling-data for WebRTC connection establishment. The protocol allows the following messages:

Message	Parameters	Semantics
handshake	peer_id, connection_type	Sent from peer to mediator to announce peer-id and the connection type (mediation or replication)
established	-	Sent from mediator to peer to announce that the connection is established
get_peers	full_hash	Sent to request peer-IDs of peers that seed the data-unit fitting the full-hash
peers	full_hash, peer_list	Sent as a response to *get_peers*
signal	full_hash, peer_id, signal_data	Sent to transport signaling information between two peers
announce	full_hash	Sent from peer to mediator, to announce the start of seeding
finish	full_hash	Sent from peer to mediator, to announce the end of seeding

Table 3.1.: Mediation Protocol Definition.

This protocol does not include DHT messages. The DHT will use its own protocol like KRPC (Kademlia). It also does not include the transport of payload-data, this is handled by a peer-wire protocol like the peer-protocol used in BitTorrent.

The following sequence describes how mediation proceeds in a conceptual manner.

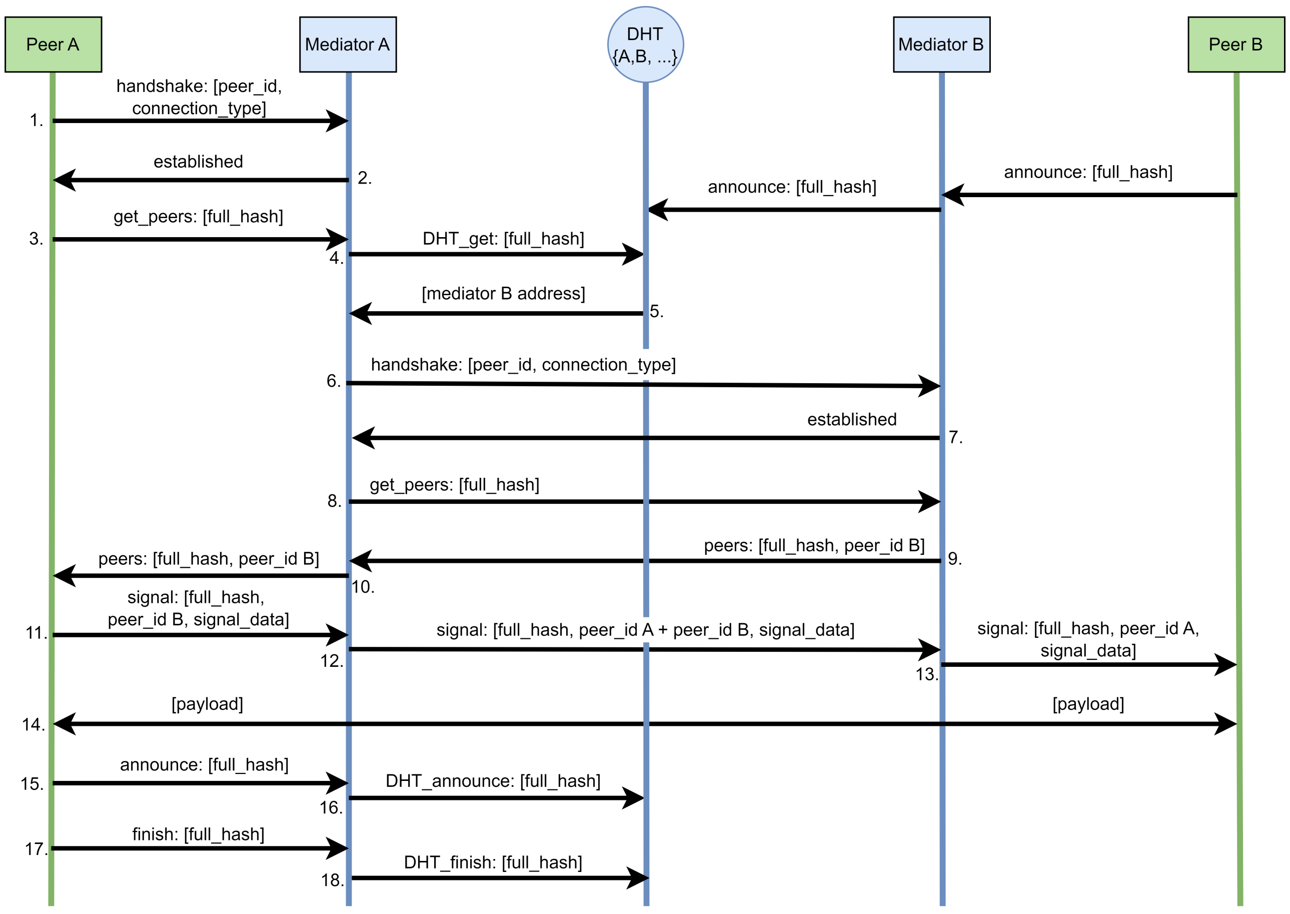

Peer A
Mediator A
DHT {A,B, ...}
Mediator B
Peer B
1. handshake: [peer_id, connection_type]
2. established
announce: [full_hash]
announce: [full_hash]
3. get_peers: [full_hash]
4. DHT_get: [full_hash]
5. [mediator B address]
6. handshake: [peer_id, connection_type]
7. established
8. get_peers: [full_hash]
9. peers: [full_hash, peer_id B]
10. peers: [full_hash, peer_id B]
11. signal: [full_hash, peer_id B, signal_data]
12. signal: [full_hash, peer_id A + peer_id B, signal_data]
13. signal: [full_hash, peer_id A, signal_data]
14. [payload]
[payload]
15. announce: [full_hash]
16. DHT_announce: [full_hash]
17. finish: [full_hash]
18. DHT_finish: [full_hash]

Figure 3.4.: Sequence Diagram of the Mediation Process.

1. Peer A sends a handshake with its peer-identifier to mediator A.
2. Mediator A confirms connection establishment by sending *established*.
3. Peer A requests a list of peers from its mediator, by a full-hash of wanted data.
4. Mediator A looks up other mediators on the DHT, by the full-hash.
5. A DHT-Node responds with contact information of mediator B, who knows of peer B.
6. Mediator A sends a handshake with its peer-identifier to mediator B.
7. Mediator B confirms connection establishment by sending *established*.
8. Mediator A requests a list of peers from mediator B.
9. Mediator B responds with the peer IDs of seeding peers it knows of.
10. Mediator A sends the list of retrieved peer-IDs to peer A.
11. Peer A picks peers it wants to connect to from the received list. It now sends signaling information for peer B to mediator A.
12. Mediator A forwards the received signaling information to mediator B. He encodes sender and receiver peer-identifiers by concatenating them.
13. Mediator B forwards the information to peer B. He only sends the senders peer-identifier after decoding the combined identifier he received before.
14. A P2P connection is now established.
15. Peer A may now announce that it has pieces of a certain data-unit, by its hash.
16. Mediator A announces to the DHT, that it knows of peers seeding pieces of certain data-units.
17. Peer A sends *finish* if he stops seeding the file.
18. Mediator A sends *finish* to the DHT to remove the peer-ID from the DHT.

Preceding this, peer B announced its seeding of a data-unit in the same manner as peer A did in steps 11 and 12. If mediator A knows enough peers to serve a peer joining the swarm, it may omit all steps that communicate with the DHT, mediator B and peer B. It must still announce that it knows of peers, seeding data, to support other nodes.

As an alternative to this approach, the client A could directly contact mediator B, to reduce latency while signaling. However, this would require more labor done by the clients. In this scenario, peer A would directly contact mediator B, after mediator A communicated contact information of mediator B to it.

3.3.2. Mediation Protocol Dialog Combinations

The dialog combinations below are possible for mediation protocol.

Message	Peer-to-Mediation	Mediation-to-Peer	Mediation-to-Mediation
handshake	x	-	x
established	-	x	x
get_peers	x	-	x
peers	-	x	x
signal	x	x	x
announce	x	-	-
finish	x	-	-

Table 3.2.: Mediation Protocol Dialog Combinations.

3.4. Security Considerations

This section acts as a threat model of the most obvious threats to the system described in this chapter. Seen from the perspective of the STRIDE threat-model, the threats listed below lay in the areas of tampering and denial of service (DoS). It is important to note that if a web application itself was malicious, this cannot be prevented by the system described since the web application delivers the necessary components of the system described to the peers. Therefore, threats of this scenario are not included in this analysis.

3.4.1. Sending Malicious Pieces (Tampering)

Component	Peer
Threat	A malicious peer sends incorrect pieces to downloading peers.
Scope	The scope of this attack is limited to single-files that the malicious peer is wrongfully seeding.
Mitigation	The meta-information provided must contain a hash of the full-file (can be a merkle-root) or in a best-case hashes for each piece. Peers must discard files and pieces of which the hash does not match with the one provided in the meta-information. Files that are not yet verified must be handled with care under the assumption that they may contain malicious code.

Table 3.3.: Malicious Piece-Sending Threat Description.

3.4.2. Hosting Malicious User Content (Tampering)

Component	Peer
Threat	A user uploads malicious content on a site which allows dynamic user content.
Scope	This attack is limited to all peers / users downloading malicious files.
Mitigation	One mitigation is to use an antivirus scanner. Another one is to use a malware voting service on which contaminated hashes can be flagged as malware.

Table 3.4.: Hosting of Malicious Files Threat Description.

3.4.3. Sybil Attacks (DoS)

Component	DHT
Threat	A malicious node operates as many identities as possible to flood the system with wrong or malicious peer-information.
Scope	The scope of this type of attack is limited by the fact that mediators do not have to rely on other mediators as long as they know enough peers themselves. Mediators of small-scale web applications might not be able to serve usable peer-information to its own peers.
Mitigation	Since it makes sense to host a mediator per web applications, mediator addresses can be constrained to second-level DNS-domains. This would not require any additional resources for legitimate hosts. A malicious node trying to create as many identities on the DHT as possible would need to buy a DNS-domain for every identity it tries to sign up, leading to great cost.

Table 3.5.: Sybil Attack Threat Description.

3.4.4. Flooding a Mediator with Malicious or Incorrect Peers (DoS)

Component	Peer, Mediator
Threat	An attacker creates malicious peers and announces them to a mediator, to prevent truthful peers from downloading content.
Scope	This attack concerns all peers connected to the attacked mediator and all files provided by the swarms advertised by that mediator.
Mitigation	Mediation for this can be implemented by a challenge response mechanism for the mediator in question to make sure that an announcing peer actually seeds the pieces it claims to have. To challenge a peer, the mediator would have to act as a peer himself. This would significantly increase the workload on the mediator. Alternatively, the mediator could challenge the peer for information that it can only know from the web application that it claims to use. The web application in-turn would only provide the challenged information after confirming that the peer is not bot-operated. While the latter solution seems more effective it could impair the user-experience for truthful users, depending on the "anti-bot" mechanism used. An example for such a mechanism would be a Completely Automated Public Turing Test (or CAPTCHA).

Table 3.6.: Flooding Malicious Peers Threat Description.

Part II.

Prototype Documentation

4. Requirements

4.1. Functional Requirements

In this section the functional requirements that the system under development must, should or could meet are described. A user utilizing a web application which is using the system based on this project is called an *end-user*. The computer system the end-user is using to visit the web application is referred to as his system.

No.	Level	Requirement
1	Must	The system must be able to let end-users download shared files to his system.
2	Must	The system must be able to let end-users share files.
3	Should	The system should be able to allow a web application to embed downloaded media content in its pages.
4	Could	The system could be able to allow a web application to embed downloaded JavaScript scripts in its pages and run them.
5	Could	A list of WebRTC-ICE candidates is configurable by the publisher of the web application.
6	Could	The system lets developers experiment with different strategies to prioritize peers and mediators.
7	Could	Peers cache downloaded files and re-seed them when re-visiting a web application

Table 4.1.: Declaration of Functional Requirements for the Prototype System under Development.

4.2. Non-Functional Requirements

This section describes non-functional requirements that the system under development must, should or could meet. Please note, since this a prototype, the NFRs do not take performance into consideration as much as a non-prototype system would likely need.

No.	Level	Requirement
1	Must	All payload transfers are implemented in a P2P manner between web applications.
2	Must	WebRTC signaling is implemented in a decentralized manner.
3	Must	Peer discovery is implemented in a decentralized manner.
4	Should	WebRTC-ICE candidates are mediated between peers.
5	Could	Peers are prioritized by lowest-latency and/or geographical closeness

Table 4.2.: Declaration of Non-Functional Requirements for the Prototype System under Development.

4.3. Constraints

The main constraint of the system under development is that the peer is running inside a browser-environment. These limits used languages to either JavaScript, TypeScript or anything compilable to WebAssembly. Connections between the peer and the mediator are limited to either WebSockets or HTTP. Connections between Peers are limited to WebRTC connections as mentioned in preceding chapters.

4.4. Context and Scope

This project is not meant to implement WebRTC itself. For WebRTC connections, libraries are used. These libraries also handle all communication with ICE-Services. They are therefore external dependencies to our system. For the DHT, we also use the existing implementation used in WebTorrent. While the BitTorrent protocol-messages are handled by the WebTorrent/BitTorrent library, the semantics of the protocol is implemented by ourselves. The Mediation protocol is completely implemented by us and is transported using the Socket.io library [27].

5. Architecture

5.1. Containers and Context

Below the details about the main containers (separately deployable units in this project) and with whom they interact within and outside the system under development are provided.

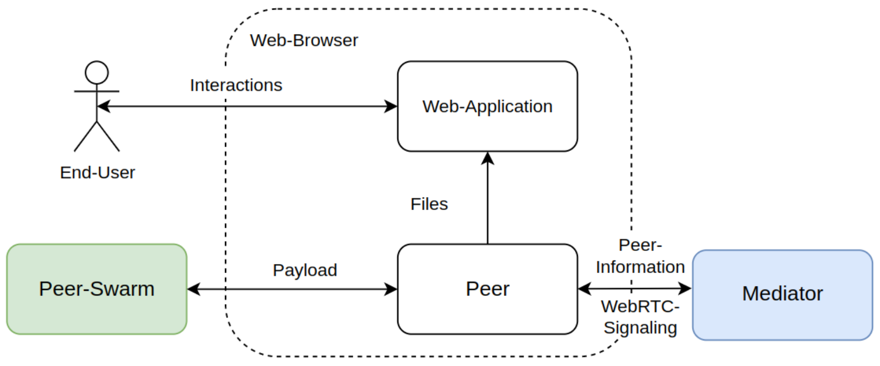


Figure 5.1.: Context of the Peer-Tier.

The peer container runs within a browser environment together with the application using it. An end-user utilizing the web application interacts with the system via the application either implicitly by using libraries and media downloaded by the system or explicitly by starting a download. The application connects to the peer swarm which consists of other peers of the same nature. The peer also connects to one mediator, which provides peer information (IDs of peers in the swarm) and WebRTC signaling information needed to establish a connection.

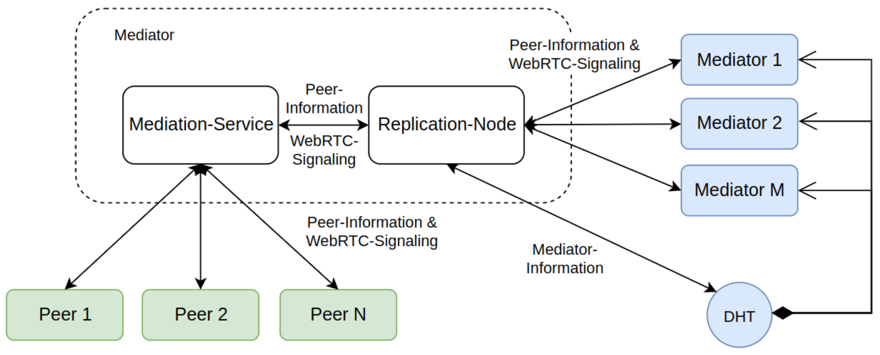


Figure 5.2.: Context of the Mediator-Tier.

The mediator container consists of two groups of functionality. The mediation service connects to peers to which it provides peer information (IDs of other peers in the swarm) and WebRTC signaling information. The replication node connects to other mediators, from which and to which it provides the peer information that is later provided to peers for connection establishment. Other mediators are located by querying the DHT, which consists of all mediators globally.

5.2. Components

The components which the system under development consists of are declared in the following part. The components shown below do not contain all the classes and modules of the source code exactly but represent the most important units of functionality and their interactions.

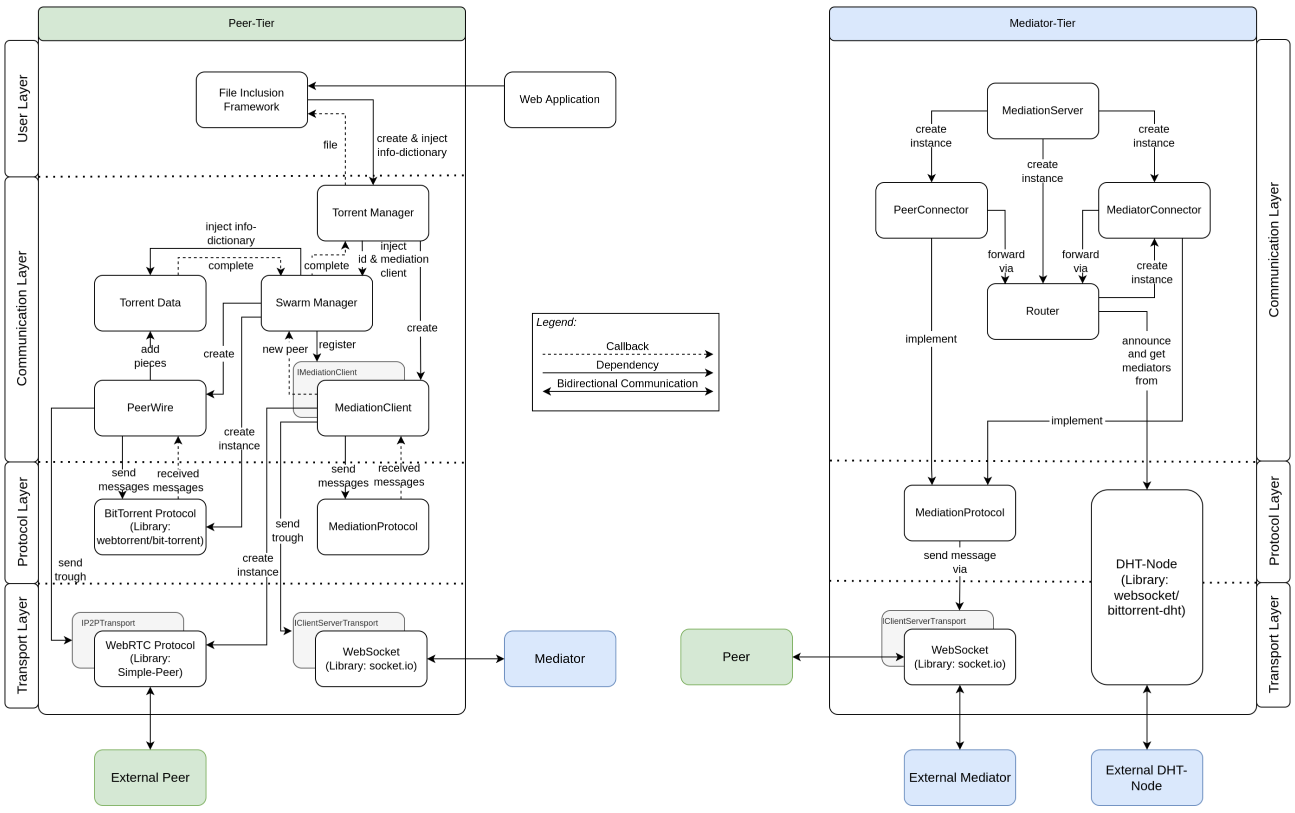


Figure 5.3.: Components Grouped into Tiers and Layers.

For our prototype we chose to group the components into layers and tiers. While the tiers classically separate the application by its physical deployment location, the layers were chosen so that they resemble the layered architecture of OSI-layered networks. In short this means that every layer N provides a service to any layer N + 1, where higher numbered layers appear above lower numbered ones in the diagram. For simplicity reasons a service of layer N is accessible to any layer above and not just its direct upper neighbor.

5.2.1. User Layer

The user as seen from this project are web developers who want to use the system built during this project in an application. The user layer, as its name suggests, provides the functionality needed to support this.

5.2.2. File Inclusion Framework

This components task is to instruct the torrent manager to download a file, by giving its info dictionaries containing meta information about the files it must download. Upon receiving these files it will provide them to the web application in a manner that is easy for the developers of the application to include them. The file inclusion framework will also apply caching via a library called "localforage" [28] if the developer tells it to do so.

5.2.3. Communication Layer

The communication layer is where semantic meaning is given to the protocols in lower layers.

5.2.4. Torrent Manager

The *Torrent Manager* takes care of all torrents that must be downloaded. It creates *Swarm Managers* for each file to download and gives them a means to find new peers by providing access to the *Mediation Client*. It also generates a single peer ID that is used by all *Swarm Managers*.

5.2.5. Swarm Manager

There is one *Swarm Manager* for each file to download. It receives new peers from the *Mediation Client* and creates new *Peer Wires* accordingly. It also initializes a *Torrent Data* object with meta information and waits for it to emit a *complete* event via a callback to forward it to the *Torrent Manager*.

5.2.6. Torrent Data

The *Torrent Data* component keeps track of all pieces of a file. It allows *Peer Wires* to acquire a piece to download so that multiple *Peer Wires* do not download the same piece.

5.2.7. Peer Wire

Peer Wire implements the semantic part of our BitTorrent implementation and encapsulates a single connection to another peer. It receives an RPC like interface from the *BitTorrent Protocol* library. The component receives information about what pieces to download from its assigned *Torrent Data* object.

5.2.8. Mediation Client

The *Mediation Client* is responsible for communicating with a *Mediator* and providing received peer- and signaling-information to the *Swarm Manager*. To communicate to *Mediators* it uses the *Mediation Protocol* described in this document via a *WebSocket* connection.

5.2.9. Protocol Layer

The protocol layer contains transport agnostic implementations of the *BitTorrent Protocol* (implemented by the web-torrent/bittorrent) library and the *Mediation Protocol* (implemented during this project).

5.2.10. BitTorrent Protocol

The *webtorrent-bittorrent* library implements all messages required by the BitTorrent protocol standard [1].

5.2.11. Mediation Protocol

Used in both tiers, the *Mediation Protocol* component encapsulates the functionality (not the semantic-meaning) of the mediation protocol. It allows other modules to register listeners to certain messages and send messages.

5.2.12. Transport Layer

The transport layer contains the components needed to transport the protocols of the upper layers to other peers and mediators.

5.2.13. WebRTC Protocol

This component is implemented by the *simple-peer* [29] library also used by the *WebTorrent* project [4]. WebRTC is used to connect to other peers.

5.2.14. WebSocket

WebSockets are used to connect to mediators, since they can be reached in a client-server fashion. Using the *socket.io* library for this seems like a reasonable choice, since it is widely used in big projects and time-tested. This component is used in both tiers.

5.2.15. Mediation Server

The *Mediation Server* accepts incoming connections, determines if they are initiated by another mediator or a peer and forwards further action to either a *PeerConnector* or a *MediatorConnector*.

5.2.16. PeerConnector and MediatorConnector

Both, the *PeerConnector* and the *MediatorConnector* implement the semantics of the *MediationProtocol*. An instance of one of these components represents a connection to a peer or a mediator. To enable them to forward messages to different peers and mediators, all instances of these components depend on a common *router*-component.

5.2.17. Router

The *Router* keeps track of all known peers and is able to forward messages to any peer or connected mediator.

5.2.18. DHT-Node

Using the "bittorrent-dht" library provided by the WebTorrent project, the DHT-Node component allows the mediator to announce that he knows of clients, seeding a certain file.

5.3. Architectural Decisions

This section describes the most important decisions we made during development of the prototype implementation of our concept. Please note that most of the more technical decisions were already anticipated in the concept.

5.3.1. Languages and Platform

For the development of the peer application, TypeScript was used. TypeScript was chosen because it allows type-safe programming. Know-how in the team was also considered in this choice. Another factor was that most libraries we wanted to use are implemented in JavaScript which is interoperable with TypeScript.

The mediator application was developed for the Node.js platform in TypeScript as well for the same reasons.

5.3.2. Libraries Used

The following libraries were used to implement this project:

Simple-Peer [29] Is a WebRTC abstraction layer that makes it easy to establish peer-to-peer connections. For our purposes it was ideal, since it does not take care of signaling but simply returns us a signaling string that can be transported to the opposite peer however we want to.

BitTorrent-Protocol [9] Is the BitTorrent implementation used by WebTorrent [4]. It was chosen because it is the most stable implementation of BitTorrent implemented in web-compatible languages.

LocalForage [28] Is a library that simplifies the API of IndexedDB [26]. It has a well-maintained repository and allows us to implement the caching features in a timely manner.

Socket.io [27] Is a well-known standard for JavaScript web-sockets. It is used by this prototype to connect peers to mediators and mediators to other mediators.

Bittorrent-DHT [30] Is the "Mainline DHT" implementation of the WebTorrent project. It is well maintained and easy to use and was therefore chosen for this implementation.

5.3.3. Implementation Details

Peer-ID generation is implemented as a factory-method, to allow for easy testability and for allowing experiments, since this is a major requirement to optimize latency according to our concept.

All the currently downloaded pieces are stored in a "TorrentData" object. TorrentData also acts as a queue for PeerWire instances to reserve and get the next piece-index that needs to be downloaded. The piece-indexes are reserved for a specific time, after which the TorrentData instance assumes that the PeerWire which originally reserved the index has failed and starts allowing PeerWires to reserve it again. Even tough this will lead to multiple PeerWires downloading the same piece in odd cases, this seemed like the appropriate way to do it because of the asynchronous nature of the whole system.

PeerWires get and reserve piece-ids from the TorrentData instance and download them sequentially. The whole torrent itself is not necessarily downloaded sequentially, since multiple PeerWire instances operate in one swarm, managed by a SwarmManager object. Even though torrents are not downloaded in a perfect sequential manner, the TorrentData object still returns piece-indexes as sequentially as possible. Although this is not a requirement in our case, it could be useful if video streams were added to this system in the future.

As proposed in our concept, mediators share peer-information directly instead of putting them onto the DHT. This makes the DHT more light-weight since it only contains information about mediators. It also increases performance, since mediators, which once they know each other, can directly request new peers from each other and share signaling-data.

The mediator does not cache "PEERS" messages from mediators because the peer-swarms experience high churn. If we allowed mediators to cache "PEERS" information from other mediators, this cache would mostly consist of deprecated information about peers that are already disconnected. This should not have too much of a performance impact on the system because external peers are a secondary measure. In most cases the mediator will know at least a few peers personally.

Our implementation does not provide functionality for a fallback HTTP-download from the web-server if no peers are available. Instead we propose for hosts of web-services to always keep one peer running on their own. This limitation exists because of time-constraints of the project.

5.4. Project Setup

The project is structured into 4 main folders.

- "mediator": contains the implementation of the mediator in the form of an NPM-project.
- "peer": contains the implementation of the peer in the form of an NPM-project.
- "common": contains source-code common to both "mediator" and "peer" projects.
- "preview": contains a preview web-site consisting of a single "index.html" file.

In the root-folder of the project, one can also find the files "compile.ps1" for PowerShell and "compile.sh" for Bash to compile all TypeScript code to JavaScript and bundle the peer into a single browser-ready file. To compile the project TypeScript, Browserify and Esmify have to be installed on the compiling system. Furthermore, to run the mediator, Node.js has to be installed.

5.4.1. Mediator Execution

To run the mediator one can simply run the command:

node mediator/out/mediator/src/index.js MEDIATOR_PORT DHT_PORT DHT_BOOTSTRAPPER

The parameters are:

- *MEDIATOR_PORT* refers to the port that the mediator will listen to requests.
- *DHT_PORT* refers to the port that the DHT-node of the mediator will listen to.
- *DHT_BOOTSTRAPPER* refers to the address and port of the bootstrapping node that is contacted to build the routing-table. It may be omitted if it is assumed that there currently exists no network using our system.

6. Quality Assessment

6.1. Testing

To test the implemented system we employed both integration tests and end-to-end tests. Automated integration tests are only applied to the mediator part of the system, because of the in-browser nature of the peer-implementation. End-to-end peer testing is done manually.

6.1.1. End-to-End Tests

The following end-to-end tests were performed by uploading and downloading different files through the developed system and analyzing logs.

Test 1: Single Mediator, same Network

Test setup	One mediator and two clients are running in the same network. Both clients are configured to connect to mentioned mediator.
Success criterion	Client 1 is able to generate meta-data for a file and seed it, client 2 is able to download that file by inputting the meta-data. The mediator answers peer-requests correctly and forwards signaling-data.
Result	Successful

Table 6.1.: Test 1: Single Mediator, same Network Test Description.

Test 2: Two Mediators, same Network

Test setup	Two mediators and two clients are running in the same network. Clients are configured so that both connect to another mediator. One mediator is configured to connect to the other one as the DHT bootstrap-node.
Success criterion	Client 1 is able to generate meta-data for a file and seed it, client 2 is able to download that file by inputting the meta-data. The mediators exchange peer information, answer peer-requests and forward signaling-data correctly.
Result	Successful

Table 6.2.: Test 2: Two Mediators, same Network Test Description.

Test 3: Single Mediator, different Networks

Test setup	One mediator and two clients are all running in different networks. Both clients run behind NAT firewalls. The mediators' firewall has port-forwarding configured to allow for incoming connections.
Success criterion	Client 1 is able to generate meta-data for a file and seed it, client 2 is able to download that file by inputting the meta-data. The WebRTC library cooperates with STUN and TURN servers to solve NAT problems.
Result	Successful

Table 6.3.: Test 3: Single Mediator, different Networks Test Description.

Test 4: Two Mediators, different Networks

Test setup	Two mediators and two clients are all running in different networks. Both clients run behind a NAT firewall. The mediators' firewall have port-forwarding configured to allow for incoming connections, both for mediation and for DHT replication purposes.
Success criterion	Client 1 is able to generate meta-data for a file and seed it, client 2 is able to download that file by inputting the meta-data. NAT is traversed successfully and peer information is handled and forwarded correctly.
Result	Successful

Table 6.4.: Test 4: Two Mediators, different Networks Test Description.

6.1.2. Integration Tests

Integration tests are created to test the basic functionalities of the mediator, as well as the communication between two mediators. The second mentioned test group implicitly verify the functionality of the DHT, because each mediator provides its own DHT node, which are required to lookup searched information.

Test Setup

The Mocha test framework [31] is used for the integration tests. It is supported by the Chai assertion library [32] to enable assertions.

At the beginning of the tests, a new instance of the test infrastructure is created. This includes a mediator (or two) running on localhost and two mediation protocol instances which are connected over sockets to a mediator.

All tests are always executed two times. In the first run only one mediator is started, and all communication occurs over it. In the second run another mediator is started, and the communication between the two mediators will be tested as well.

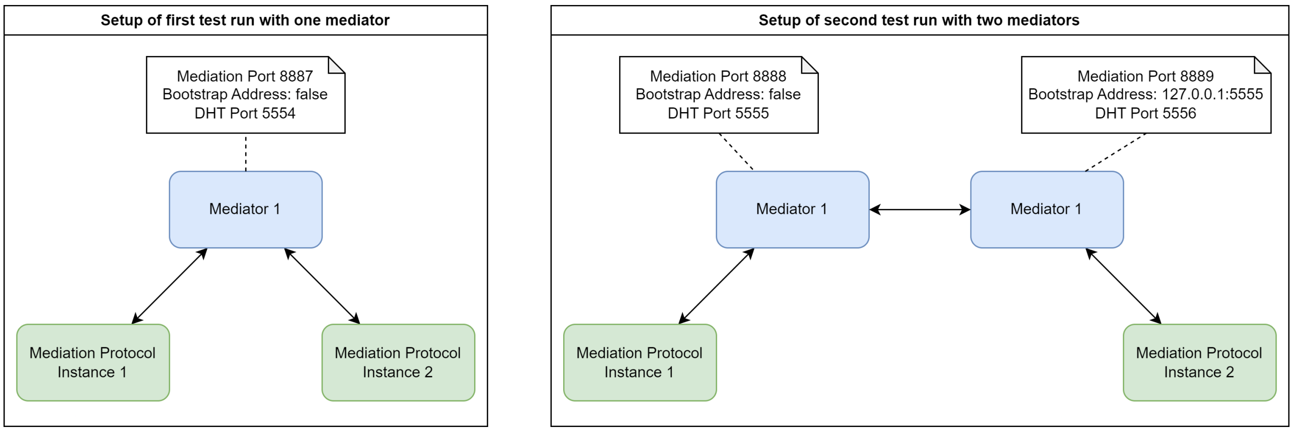

Setup of first test run with one mediator
Mediation Port 8887
Bootstrap Address: false
DHT Port 5554
Mediator 1
Mediation Protocol Instance 1
Mediation Protocol Instance 2
Setup of second test run with two mediators
Mediation Port 8888
Bootstrap Address: false
DHT Port 5555
Mediation Port 8889
Bootstrap Address: 127.0.0.1:5555
DHT Port 5556
Mediator 1
Mediator 1
Mediation Protocol Instance 1
Mediation Protocol Instance 2

Figure 6.1.: Test Setup of Integration Tests.

Mediator Tests

Due to network transmissions over the mediation protocol, all tests must be performed in the designated order.

No.	Test Title	Result one Mediator	Result two Mediators
01	sendHandshake_noError	Successful	Successful
02	isEstablished_true	Successful	Successful
03	getPeersBeforeAnnounce_noError	Successful	Successful
04	peers_emptyResult	Successful	Successful
05	peers_consoleWarn	Successful	Successful
06	announce_noError	Successful	Successful
07	getPeersAfterAnnounce_noError	Successful	Successful
08	peers_oneEntry	Successful	Successful
09	signal_noError	Successful	Successful
10	signalBack_noError	Successful	Successful
11	signal_receiveResult	Successful	Successful
12	finish_noError	Successful	Successful
13	getPeersAfterFinish_noError	Successful	Successful
14	peersAfterFinish_emptyResult	Successful	Successful

Table 6.5.: Test Protocol of the Integration Tests.

6.2. Requirements Analysis

6.2.1. Functional Requirements

No.	Level of Fulfillment
1	Fulfilled by functionality of the prototype.
2	Fulfilled implicitly. A user shares files whenever he downloads them.
3	Fulfilled by functionality of the prototype.
4	Currently unsupported due to lack of time.
5	Fulfilled by functionality of the prototype.
6	Fulfilled partially. There is no explicit to do so but the modular nature of the prototype allows for easy modifications of prioritization.
7	Fulfilled by functionality of the prototype.

Table 6.6.: Analysis of Functional Requirements.

6.2.2. Non-Functional Requirements

No.	Level of Fulfillment
1	Fulfilled. Payloads are transferred P2P, except for cases when one or both peers reside behind a symmetric NAT. In this case STUN and TURN services are used additionally.
2	Fulfilled. WebRTC signaling is done decentralized via the mediators, the clients chose to connect to.
3	Fulfilled. Peer discovery is done decentralized via the mediators as well.
4	Fulfilled. ICE candidates are mediated between peers. This is implemented in the ”Simple-Peer” library we used.
5	Not fulfilled. Peers are currently not prioritized by any fixed metric.

Table 6.7.: Analysis of Non-Functional Requirements.

6.3. Performance Analysis

This section contains a qualitative performance analysis of our system. The following tests are performed via the internet. For our tests we used ICE-candidates hosted by the Openrelay-Project [33].

6.3.1. Test Setup

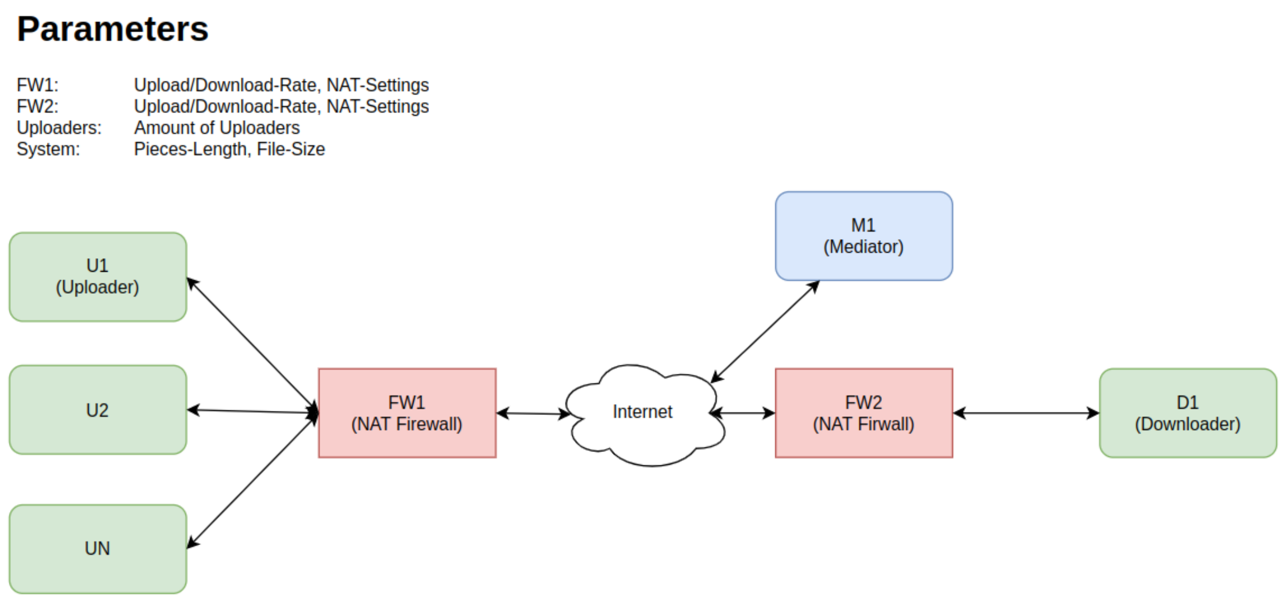

Parameters
FW1: Upload/Download-Rate, NAT-Settings
FW2: Upload/Download-Rate, NAT-Settings
Uploaders: Amount of Uploaders
System: Pieces-Length, File-Size
U1 (Uploader)
U2
UN
FW1 (NAT Firewall)
Internet
M1 (Mediator)
FW2 (NAT Firwall)
D1 (Downloader)

Figure 6.2.: Performance Analysis Test-Setup.

Both the network enclosing the uploaders or seeders and the network enclosing the downloader or leecher lay behind a firewall/router with symmetric NAT enabled. The clients are configured to use STUN and/or TURN to establish connections. The mediator is in the same network as the downloader but is exempt from any firewall/NAT rules. Both firewalls are standard devices meant for private use.

Time measurements given in the following tests include connection-establishment and correctness-assertions done by the system.

6.3.2. Time vs Pieces-Length

This test analyzes the time needed to download a file with different pieces sizes.

Test Configuration

FW1	Download-Rate: 50 MB/s, Upload-Rate: 30 MB/s, NAT-Type: Symmetric
FW2	Download-Rate: 100 MB/s, Upload-Rate: 30 MB/s, NAT-Type: Symmetric
Uploaders	Amount: 1
System	Pieces-Length: 1 KB, 10 KB, 100 KB, 150 KB, 200 KB, File-Size: 10 MB

Table 6.8.: Time vs Pieces-Length Test Configuration.

Only one uploader is used to isolate the effect of different pieces lengths on performance.

Results

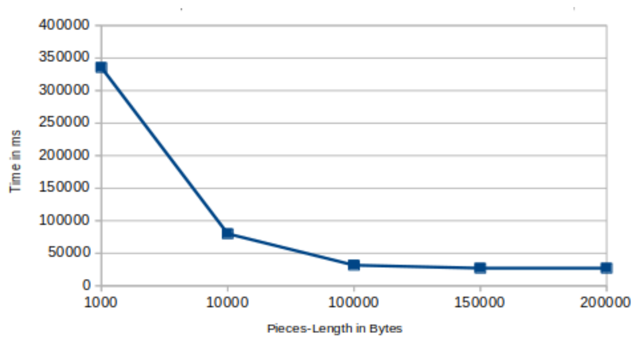


Figure 6.3.: Time vs Pieces-Length Test Results.

The results clearly show a logarithmic tendency of the relation between performance and pieces-length. Any improvements above 200 KB would be insignificant. This also makes the limitation of pieces-length of about 244 KB laid upon this system by the libraries in use insignificant. The plot shows the average amount of time from two measurements.

6.3.3. Time vs Uploaders-Amount

This test analyzes the time needed to download a file with different amounts of uploaders.

Test Configuration

FW1	Download-Rate: 50 MB/s, Upload-Rate: 30 MB/s, NAT-Type: Symmetric
FW2	Download-Rate: 100 MB/s, Upload-Rate: 30 MB/s, NAT-Type: Symmetric
Uploaders	Amount: 1-8
System	Pieces-Length: 200 KB, File-Size: 10 MB

Table 6.9.: Time vs Uploaders-Amount Test Configuration.

Pieces-length and file-size are fixed to their respective values to isolate the effect of different amounts of seeders.

Results

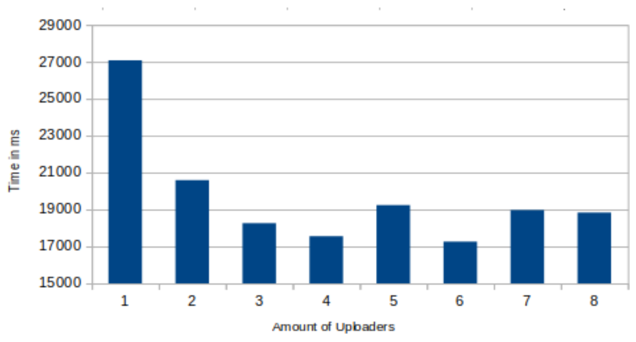


Figure 6.4.: Time vs Uploaders-Amount Test Results.

Even though times improve from one seeder to two, the relation seems almost constant after that. This is likely due to the fact that this test was conducted in a stable environment without peers-leaving during downloads. We estimate that the effect in a more volatile environment would drastically change.

6.3.4. File-Size vs Throughput

This test analyzes the throughput achieved with different file-sizes.

Test Configuration

FW1	Download-Rate: 300 MB/s, Upload-Rate: 65 MB/s, NAT-Type: Symmetric
FW2	Download-Rate: 100 MB/s, Upload-Rate: 30 MB/s, NAT-Type: Symmetric
Uploaders	Amount: 1
System	Pieces-Length (in KB): 200, File-Size: 10 MB, 100 MB, 1000 MB

Table 6.10.: File-Size vs Throughput Test Configuration.

The amount-of uploaders and pieces length are fixed to isolate the effect of different file-sizes on throughput.

Results

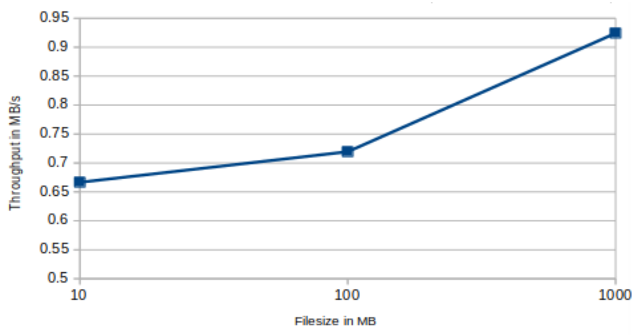


Figure 6.5.: File-Size vs Throughput Test Results.

The throughput increases with bigger files. This is because connection establishment was included in the measurement. The time needed for connection-establishment becomes more insignificant with larger files. Adjusted for the X-axis being exponential the plot above can be interpreted as approximately linear.

6.4. Technical Debt

This section describes the shortcuts we had to take during development because of time constraints.

6.4.1. Peer Wire Implementation

The peer-wire implementation, responsible for actually downloading and seeding pieces currently answers to any request for pieces and tries to request pieces as fast as possible. In large swarms this would lead to major performance issues. To resolve this, the messages "choke", "unchoke", "interested" and "uninterested" would have to be implemented correctly and sent at the right time. This could be done by gradually analyzing the used bandwidth at runtime and slowing down requests and responses to avoid congestion.

6.4.2. Peer Prioritization

A peer currently connects to all peers that it receives from its mediator. This leads to a lot of work being done for connection-establishment. In a production environment peers would need to be prioritized and the amount of peers connected to would need to be limited to a certain value, to lower the resources needed for connection-establishment.

6.4.3. Info-Dictionary Proofs

The info-dictionary shared between peers to download files currently contains all pieces-hashes and the merkle-root to verify the correctness of a file. Since the merkle-root is included, the pieces-hashes could be omitted. Instead, a merkle-proof could be requested from the mediator when needed to reduce the info-dictionary to a file-name and the merkle-root or even just a magnet-link format.

7. Final Prototype

7.1. Mediator

The mediator is a Node.js program that both accepts peer-requests, shares them with other mediators and finds other mediators via a DHT mechanism. The mediator can run on its own on any machine. To make it available to the public, a provider has to make sure that both the port for mediation-services and the port for the DHT-mechanism are forwarded to the machine hosting the mediator.

7.2. Peer

The peer is a JavaScript library compiled from Typescript that can be used in any web application to download and upload files to the network. The generated cdn.js file exposes an object called "cdn", which contains all functions that are available to do so. By default, two STUN-servers and two TURN-servers are configured to be used, namely:

- stun:openrelay.metered.ca:80
- turn:openrelay.metered.ca:80
- stun:openrelay.metered.ca:443
- turn:openrelay.metered.ca:443

These servers are statically configurable in the "ICEConfig.ts" file or alternatively using the "overrideIceCandidates" function, described below, at runtime. The listed servers are provided by the Openrelay-Project [33] for free.

initialize(infoDictionaries, mediatorAddress, mediatorPort, enableCaching) Initializes the library with the "InfoDictionaries" that may later be used to download the files described by them.

overrideIceCandidates(candidates) Lets the user specify ICE candidates other than the ones specified by default. The candidates-array must contain elements of the form {urls:"ice-url", username: "my-username", credential: "my-password"}. The "username" and "credential" field may be omitted when specifying STUN-servers.

includeDownloads(fileNames, cssStrings, callbacks) Downloads the specified files and allows a user to save them by clicking on the elements specified by the cssStrings. When the download is complete and the file is ready to be saved, the callback of the file is called.

includeImages(fileNames, cssStrings, callbacks) Downloads the specified images and sets them as the source of the HTMLImageElement specified in cssStrings. When the download is complete and the file is ready to be saved, the callback of the image is called.

seedFile(file, mediatorAddress, mediatorPort, dictionaryCallback) Seeds a file to the network, announcing to the mediator specified and allowing the user to share the info-dictionary returned to the dictionaryCallback.

7.3. Preview

The preview web application allows its user to test the system.

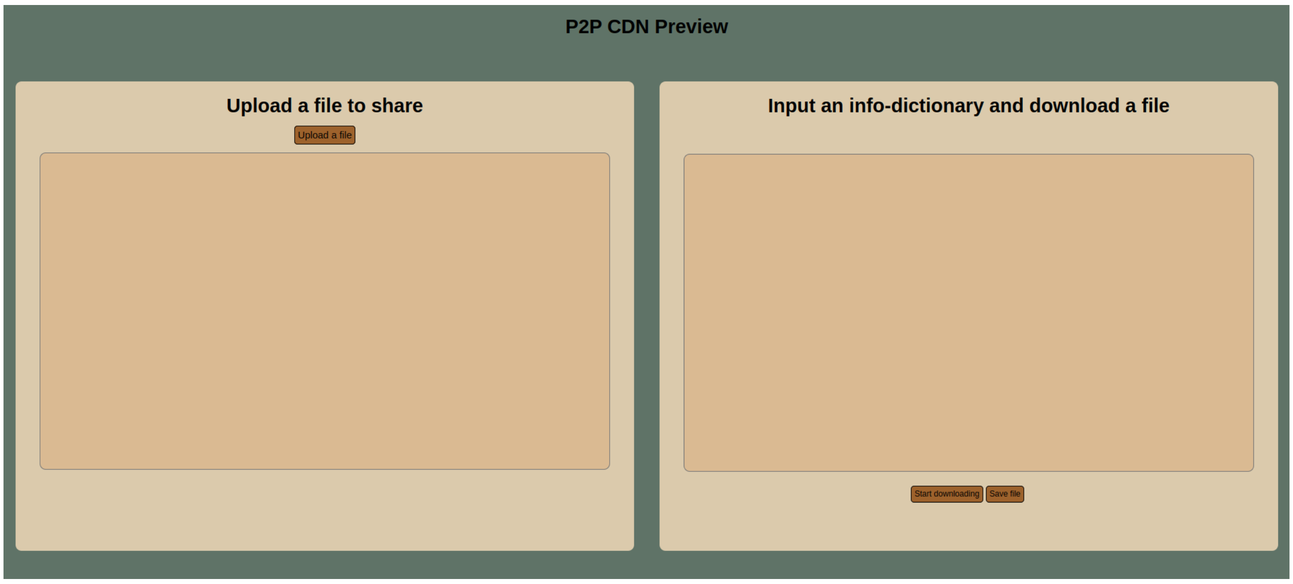
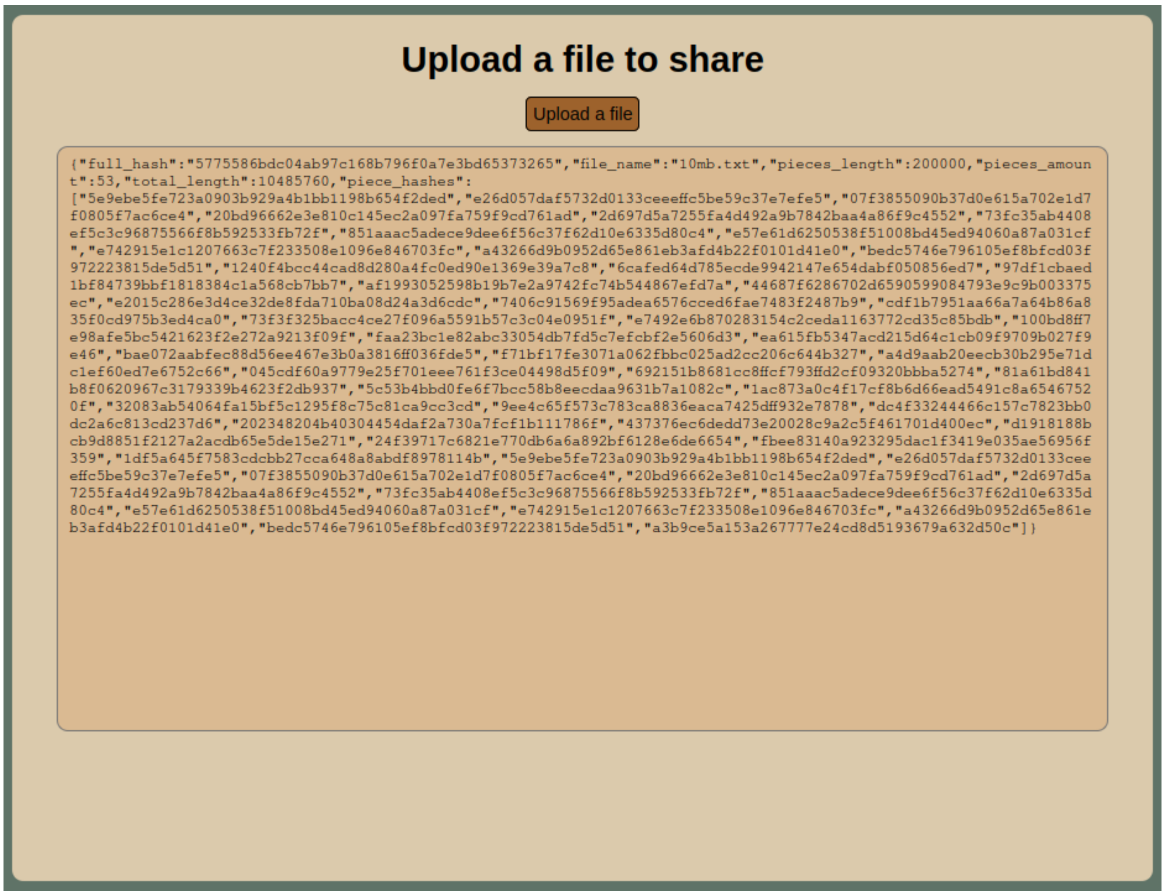


Figure 7.1.: Screenshot of the Preview Website.

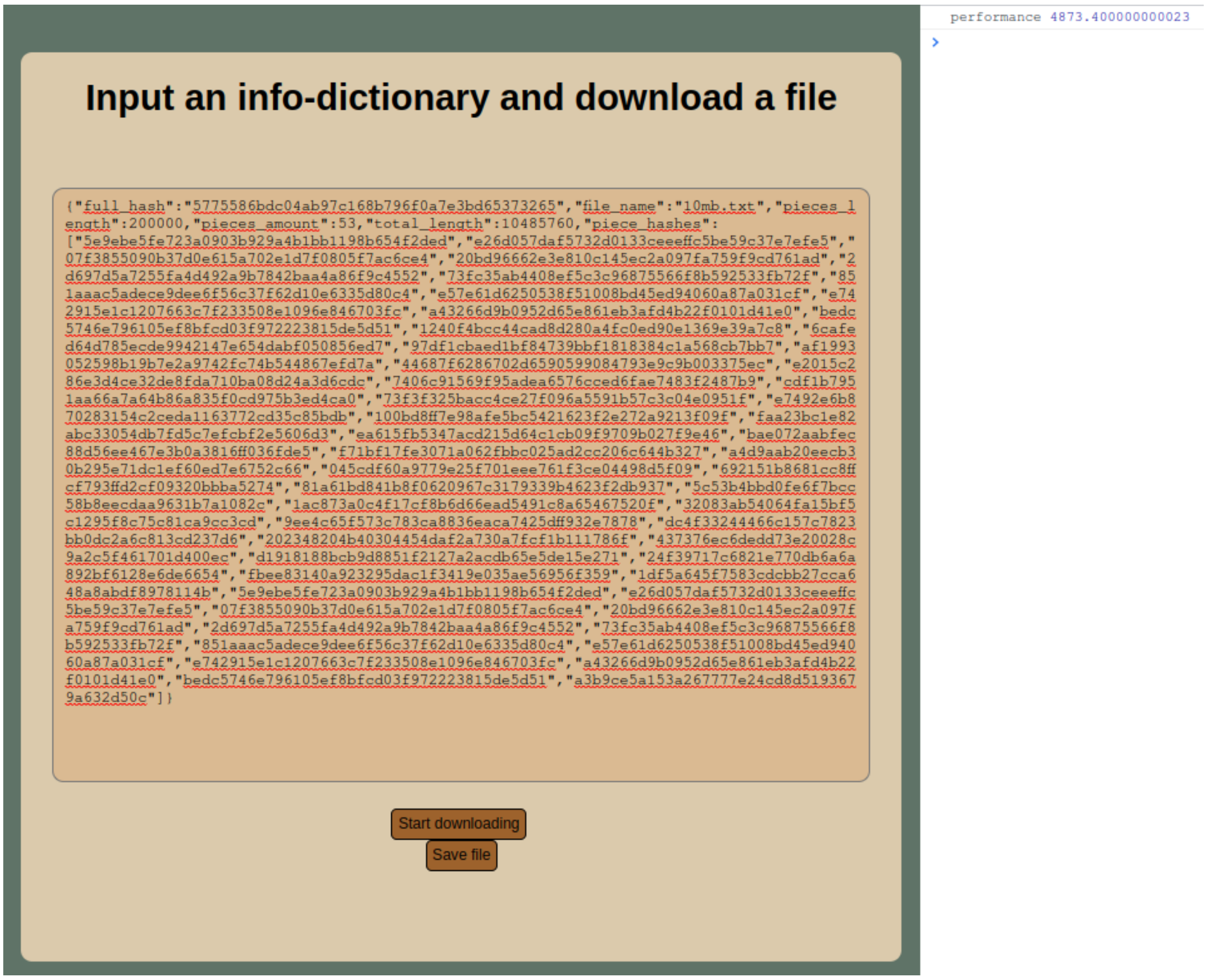
The user interface shows two boxes. One for seeding files and one for downloading and then seeding them.



```
{"full_hash":"5775586bdc04ab97c168b796f0a7e3bd65373265","file_name":"10mb.txt","pieces_length":200000,"pieces_amoun
t":53,"total_length":10485760,"piece_hashes":
["5e9ebe5fe723a0903b929a4b1bb1198b654f2ded","e26d057daf5732d0133ceeeffc5be59c37e7efe5","07f3855090b37d0e615a702e1d7
f0805f7ac6ce4","20bd96662e3e810c145ec2a097fa759f9cd761ad","2d697d5a7255fa4d492a9b7842baa4a86f9c4552","73fc35ab4408
ef5c3c96875566f8b592533fb72f","851aaac5adece9dee6f56c37f62d10e6335d80c4","e57e61d6250538f51008bd45ed94060a87a031cf
","e742915e1c1207663c7f233508e1096e846703fc","a43266d9b0952d65e861eb3afd4b22f0101d41e0","bedc5746e796105ef8bfcd03f
972223815de5d51","1240f4bcc44cad8d280a4fc0ed90e1369e39a7c8","6cafed64d785ecde9942147e654dabf050856ed7","97df1cbaed
1bf84739bbf1818384c1a568cb7bb7","af1993052598b19b7e2a9742fc74b544867efd7a","44687f6286702d6590599084793e9c9b003375
ec","e2015c286e3d4ce32de8fda710ba08d24a3d6cdc","7406c91569f95adea6576cced6fae7483f2487b9","cdf1b7951aa66a7a64b86a8
35f0cd975b3ed4ca0","73f3f325bacc4ce27f096a5591b57c3c04e0951f","e7492e6b870283154c2ceda1163772cd35c85bdb","100bd8ff7
e98afe5bc5421623f2e272a9213f09f","faa23bc1e82abc33054db7fd5c7efcbf2e5606d3","ea615fb5347acd215d64c1cb09f9709b027f9
e46","bae072aabfec88d56ee467e3b0a3816ff036fde5","f71bf17fe3071a062fbbc025ad2cc206c644b327","a4d9aab20eecb30b295e71d
c1ef60ed7e6752c66","045cdf60a9779e25f701eee761f3ce04498d5f09","692151b8681cc8ffcf793ffd2cf09320bbba5274","81a61bd841
b8f0620967c3179339b4623f2db937","5c53b4bbd0fe6f7bcc58b8eecdaa9631b7a1082c","1ac873a0c4f17cf8b6d66ead5491c8a6546752
0f","32083ab54064fa15bf5c1295f8c75c81ca9cc3cd","9ee4c65f573c783ca8836eaca7425dff932e7878","dc4f33244466c157c7823bb0
dc2a6c813cd237d6","202348204b40304454daf2a730a7fcf1b111786f","437376ec6dedd73e20028c9a2c5f461701d400ec","d1918188b
cb9d8851f2127a2acdb65e5de15e271","24f39717c6821e770db6a6a892bf6128e6de6654","fbee83140a923295dac1f3419e035ae56956f
359","1df5a645f7583cdcbb27cca648a8abdf8978114b","5e9ebe5fe723a0903b929a4b1bb1198b654f2ded","e26d057daf5732d0133cee
effc5be59c37e7efe5","07f3855090b37d0e615a702e1d7f0805f7ac6ce4","20bd96662e3e810c145ec2a097fa759f9cd761ad","2d697d5a
7255fa4d492a9b7842baa4a86f9c4552","73fc35ab4408ef5c3c96875566f8b592533fb72f","851aaac5adece9dee6f56c37f62d10e6335d
80c4","e57e61d6250538f51008bd45ed94060a87a031cf","e742915e1c1207663c7f233508e1096e846703fc","a43266d9b0952d65e861e
b3afd4b22f0101d41e0","bedc5746e796105ef8bfcd03f972223815de5d51","a3b9ce5a153a267777e24cd8d5193679a632d50c"]}
```

Figure 7.2.: Screenshot of the Preview Website with an Info-Dictionary Displayed after Seeding a File.

In the left box the user can easily start to seed a file to the network. The box below the button to choose a file will contain the info-dictionary describing the file.



```
{"full_hash":"5775586bdc04ab97c168b796f0a7e3bd65373265","file_name":"10mb.txt","pieces_l
ength":200000,"pieces_amount":53,"total_length":10485760,"piece_hashes":
["5e9ebe5fe723a0903b929a4b1bb1198b654f2ded","e26d057daf5732d0133ceeeffc5be59c37e7efe5","
07f3855090b37d0e615a702e1d7f0805f7ac6ce4","20bd96662e3e810c145ec2a097fa759f9cd761ad","2
d697d5a7255fa4d492a9b7842baa4a86f9c4552","73fc35ab4408ef5c3c96875566f8b592533fb72f","85
1aaac5adece9dee6f56c37f62d10e6335d80c4","e57e61d6250538f51008bd45ed94060a87a031cf","e74
2915e1c1207663c7f233508e1096e846703fc","a43266d9b0952d65e861eb3afd4b22f0101d41e0","bedc
5746e796105ef8bfcd03f972223815de5d51","1240f4bcc44cad8d280a4fc0ed90e1369e39a7c8","6cafe
d64d785ecde9942147e654dabf050856ed7","97df1cbaed1bf84739bbf1818384c1a568cb7bb7","af1993
052598b19b7e2a9742fc74b544867efd7a","44687f6286702d6590599084793e9c9b003375ec","e2015c2
86e3d4ce32de8fda710ba08d24a3d6cdc","7406c91569f95adea6576cced6fae7483f2487b9","cdf1b795
1aa66a7a64b86a835f0cd975b3ed4ca0","73f3f325bacc4ce27f096a5591b57c3c04e0951f","e7492e6b8
70283154c2ceda1163772cd35c85bdb","100bd8ff7e98afe5bc5421623f2e272a9213f09f","faa23bc1e82
abc33054db7fd5c7efcbf2e5606d3","ea615fb5347acd215d64c1cb09f9709b027f9e46","bae072aabfec
88d56ee467e3b0a3816ff036fde5","f71bf17fe3071a062fbbc025ad2cc206c644b327","a4d9aab20eecb3
0b295e71dc1ef60ed7e6752c66","045cdf60a9779e25f701eee761f3ce04498d5f09","692151b8681cc8ff
cf793ffd2cf09320bbba5274","81a61bd841b8f0620967c3179339b4623f2db937","5c53b4bbd0fe6f7bcc
58b8eecdaa9631b7a1082c","1ac873a0c4f17cf8b6d66ead5491c8a65467520f","32083ab54064fa15bf5
c1295f8c75c81ca9cc3cd","9ee4c65f573c783ca8836eaca7425dff932e7878","dc4f33244466c157c7823
bb0dc2a6c813cd237d6","202348204b40304454daf2a730a7fcf1b111786f","437376ec6dedd73e20028c
9a2c5f461701d400ec","d1918188bcb9d8851f2127a2acdb65e5de15e271","24f39717c6821e770db6a6a
892bf6128e6de6654","fbee83140a923295dac1f3419e035ae56956f359","1df5a645f7583cdcbb27cca6
48a8abdf8978114b","5e9ebe5fe723a0903b929a4b1bb1198b654f2ded","e26d057daf5732d0133ceeeffc
5be59c37e7efe5","07f3855090b37d0e615a702e1d7f0805f7ac6ce4","20bd96662e3e810c145ec2a097f
a759f9cd761ad","2d697d5a7255fa4d492a9b7842baa4a86f9c4552","73fc35ab4408ef5c3c96875566f8
b592533fb72f","851aaac5adece9dee6f56c37f62d10e6335d80c4","e57e61d6250538f51008bd45ed940
60a87a031cf","e742915e1c1207663c7f233508e1096e846703fc","a43266d9b0952d65e861eb3afd4b22
f0101d41e0","bedc5746e796105ef8bfcd03f972223815de5d51","a3b9ce5a153a267777e24cd8d519367
9a632d50c"]}
```

```
performance 4873.400000000023
>
```

Figure 7.3.: Screenshot of the Preview Website with Performance Measurement after Downloading a File.

The info-dictionary can be pasted into the right box on the same machine or a remote one to download the file and save it. The console output shows the time needed to download the file including finding peers and connection-establishment.

Part III.

Conclusion and Outlook

8. Conclusion and Outlook

8.1. Results

In the first part of the paper it was conceptually researched if a complete decentralized peer-to-peer CDN can be built without the user having to download a client or to install a browser extension. As a proof of concept, a web-based prototype was developed which allows us to validate that it is possible.

Our prototype is conceptually an extension of WebTorrent [4]. We used some WebTorrent libraries as a base. However, instead of relying on central trackers like WebTorrent, we have developed our own peer discovery mechanism. This is the main difference, respectively the evolution compared to WebTorrent. Therefore, the solution is fully decentralized.

The prototype has been released under MIT license on GitHub [34].

8.2. Production Readiness

Overall, we are satisfied with the result. We have successfully proven our theory. The prototype can basically be used in production in its current state. However, in order to be used meaningfully, enough peers, and ideally, multiple mediators need to utilize the network.

There is one noteworthy limitation. The performance is not comparable to modern CDNs. It is especially noticeable with small files. This is mostly because of the slow WebRTC connection establishment, especially when ICE has to be used.

In addition, the prototype could be further optimized (see Section 8.3).

8.3. Outlook

There are different topics, which were considered only conceptually due to time constraints. These could be implemented in the prototype in the future.

8.3.1. Geographical Closeness

Currently, randomly generated IDs are used. One possible optimization that we have considered in the research is the concept of geographical closeness (see Subsection 3.2.6), by using Hilbert space-filling curve [15]. The approach is to map the geographic coordinates (longitude and latitude) together with a key space to a one-dimensional space. This could possibly reduce latency by prioritizing geographically close nodes and reducing roundtrip time. However, this needs more research to develop an ideal mechanism. Approaches to this already exist in the paper about the NL-DHT [25].

8.3.2. Tampering (Security)

In Subsection 3.4.2 we discussed the security concern of tampering by a user hosting malicious content on a site which allows dynamic user content. One mitigation proposal was to use a malware voting service to flag contaminated hashes as malware. In the blockchain module at OST - Eastern Switzerland University of Applied Sciences, we created a prototype of an on-chain voting service (SmartTorrent) [35]. There we used the Goerli Testnet [36] to publish the contracts. The voting service prototype could be moved in further work to the Ethereum Mainnet [37] and could be used in our P2P-CDN to mitigate the named security concern.

8.3.3. True Web-Browser Sockets

A lot of work in this project was invested in WebRTC and its complicated connection establishment. It is the only reliable and widely used way of transport for peer-to-peer web applications. For the web to become a true application platform, a more direct way of communication will have to be implemented eventually. Today this is only possible by installing client-software outside the browser and communicating with it via browser plugins [12][13]. This is the case because of security concerns that are not solved as of today [11].

In the future, a lot of work will have to be done in this area. If this will ever be implemented, browsers could eventually take part in DHTs instead of just interacting with them indirectly. Web-peers would then no longer need any external systems for true peer-to-peer communication. This alternative to our approach could significantly change the way the internet is used. However, this is out of scope for a student research project. Further research in this field would be very interesting.

Part IV.

Appendix

9. Project Plan

This project is planned with the process framework "Ration Unified Process (RUP)". There are four project life-cycle phases:

- Inception phase: Initial research.
- Elaboration phase: Determine existing technologies and create proof of concept for possible improvements.
- Construction phase: Development of a prototype, based on the research results.
- Transition phase: Submission of the thesis and the prototype.

Rough project plan:

Week	**Start**	**Phase**	**Details**
01	19.09.2022	Inception	Project kickoff, initial research.
02	26.09.2022	Elaboration	Identify existing technologies and determine focus of the student research project.
03	03.10.2022	Elaboration	Research in the field of (geo)location based hashes and DHT optimization (for example by using the Hilbert Curve). Develop the concept.
04	10.10.2022	Elaboration, Construction	Finalizing concept draft. Start of prototype development.
05	17.10.2022	Elaboration, Construction	Further considerations with Hilbert Curve.
06	24.10.2022	Elaboration, Construction	Prototype development.
07	31.10.2022	Elaboration, Construction	Finalizing the interfaces between peer and mediator. Prototype development.
08	07.11.2022	Elaboration, Construction	Threat modelling. Prototype development.
09	14.11.2022	Construction	Prototype development: File upload and peer to peer communication over a mediator.
10	21.11.2022	Construction	Prototype development: Mediator to mediator communication.
11	28.11.2022	Construction	Prototype development: Overhaul mediator to mediator communication. Usage of DHT. Extensive Testing.
12	05.12.2022	Construction	Prototype development: Caching. Extensive Testing.
13	12.12.2022	Construction	Finalizing prototype. Expanding documentation.
14	19.12.2022	Transition	Finalizing prototype and documentation. Abstract submission (19.12.2022) Report and prototype submission (23.12.2022)

Table 9.1.: Project Plan.

10. Personal Reports

10.1. Adrian Locher

P2P protocols have always fascinated me. This project has given me the chance to learn more about the algorithms, protocols and mechanisms used to make P2P applications possible. Furthermore, I have gained new insight in the limitations of the browser and its potential as an application platform. I believe that browser-applications will eventually have to change the way in which they communicate to become more efficient and allow for mechanisms that are used in machine-native applications. The hardest challenge to do so will be to overcome the security issues that come with native networking in the web-browser.

Together with Jason Benz, working on this project was fun and coordinating the different parts of the implementation of the prototype was as easy as it can get. We managed to distribute the workload almost perfectly onto the 14 weeks we were given. The only exception to this was, that it was unexpectedly challenging to configure NAT-traversal correctly.

10.2. Jason Benz

For the student research project, it was important to me not to develop a standard business application, because I will probably be doing that often enough in the future. This topic was perfect because we got the chance to do research on a specific subject, create a concept and validate it with a prototype. Until the beginning of the project I did not have much contact with P2P applications, or the technical aspects of it. However, I find decentralized approaches very interesting, which is why I was very excited when we got the confirmation for the project. In my opinion, it does not make sense to completely turn the internet upside down and decentralize everything. But in some cases - especially with large and widely distributed files - it would definitely be worth it. It was a great pleasure to be able to do research in this field, and I was allowed to learn a lot of new things in the field of P2P, decentralization, how new protocols are built and so on.

I have already completed several projects together with Adrian Locher. As expected, the collaboration worked out excellently again, we were able to divide the workload and to supplement each other very well. During the project, some challenges arose. Together we were able to solve them. I am very satisfied with the cooperation and the result of this student research project.

11. Bibliography

[1] B. Cohen, “The BitTorrent Protocol Specification,” tech. rep., BitTorrent, 2008.

[2] “BitTorrent FAQ.” https://www.bittorrent.com/token/bittorrent-speed/faq#What-is-torrenting. Accessed: 2022-10-04.

[3] “WebRTC Wave Signaling.” https://github.com/ggerganov/wave-share. Accessed: 2022-11-01.

[4] “WebTorrent GitHub Project.” https://github.com/webtorrent. Accessed: 2022-10-04.

[5] F. Aboukhadijeh, “Dweb: Building a Resilient Web with WebTorrent,” *Mozilla Hacks*, 2018.

[6] “Hypercore Website.” https://hypercore-protocol.org/. Accessed: 2022-10-08.

[7] “IPFS Website.” https://ipfs.tech/. Accessed: 2022-10-08.

[8] “PeerCDN Acquired by Yahoo!.” https://web.archive.org/web/20150810065820/https://peercdn.com/. Accessed: 2022-10-08.

[9] F. Aboukhadijeh, “webtorrent/bittorrent-protocol.” https://github.com/webtorrent/bittorrent-protocol. Accessed: 2022-11-01.

[10] “Direct Sockets.” https://github.com/WICG/direct-sockets/blob/main/docs/explainer.md, 2022. Accessed: 2022-11-09.

[11] “Raw Sockets API.” https://github.com/mozilla/standards-positions/issues/431. Accessed: 2022-11-06.

[12] Mozilla, “Native Messaging.” https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Native_messaging. Accessed: 2022-11-07.

[13] Google, “Native Messaging.” https://developer.chrome.com/docs/apps/nativeMessaging/. Accessed: 2022-11-07.

[14] “Socketify.” https://github.com/NetAsmCom/Socketify, 2018. Accessed: 2022-11-07.

[15] D. Hilbert, “Über die stetige Abbildung einer Linie auf ein Flächenstück.,” *Mathematische Annalen*, vol. 38, 1891.

[16] O. van der Spek, “UDP Tracker Protocol for BitTorrent,” tech. rep., BitTorrent, 2008.

[17] Unknown, “Peer Exchange (PEX),” tech. rep., BitTorrent, 2015.

[18] A. N. Andrew Loewenstern, “DHT Protocol,” tech. rep., BitTorrent, 2008.

[19] D. M. Petar Maymounkov, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric,” tech. rep., New York University, 2002.

[20] Unkown, “WebTorrent Issue 288.” https://github.com/webtorrent/webtorrent/issues/288, 2015.

[21] S. Bazerque, “Hyper Hyper Space.” https://www.hyperhyperspace.org/whitepaper/, 2021.

[22] Google, “WebRTC.” https://webrtc.org. Accessed: 2022-11-01.

[23] L. R. Demian Thoma, “Distributed WebRTC Signaling,” tech. rep., Ostschweizer Fachhochschule, 2018.

[24] B. K. Raul Jimenez, Flutra Osmani, “Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay,” tech. rep., KTH Royal Institute of Technology, 2013.

[25] S. S. Saurabh Ratti, Behnoosh Hariri, “NL-DHT: A Non-uniform Locality Sensitive DHT Architecture for Massively Multi-user Virtual Environment Applications,” tech. rep., University of Ottawa, 2008.

[26] M. Contributors, “IndexedDB API MDN.” Accessed: 2022-11-06.

[27] “Socket.io.” https://socket.io/. Accessed: 2022-11-10.

[28] "LocalForage." https://localforage.github.io/localForage/. Accessed: 2022-12-01.

[29] "Simple-Peer." https://github.com/feross/simple-peer. Accessed: 2022-10-08.

[30] "Bittorrent-DHT." https://github.com/webtorrent/bittorrent-dht. Accessed: 2022-11-05.

[31] "Mocha Test Framework." https://mochajs.org. Accessed: 2022-12-05.

[32] "Chai Assertion Library." https://www.chaijs.com/. Accessed: 2022-12-05.

[33] https://www.metered.ca/tools/openrelay/. Accessed: 2022-12-16.

[34] "P2P-CDN Code Repository." https://github.com/Peer-to-Peer-CDN/P2P-CDN, Accessed: 2022-12-21.

[35] "SmartTorrent." https://github.com/Peer-to-Peer-CDN/SmartTorrent. Accessed: 2022-12-20.

[36] "Goerli Testnet." https://goerli.net/. Accessed: 2022-12-12.

[37] "Ethereum Mainnet." https://ethereum.org/en/enterprise/. Accesed: 2022-12-12.

12. List of Figures

13. List of Tables