Distributed Client Driven Certificate Transparency Log

Robin Ellgren
Tobias Löfgren

Supervisor: Niklas Carlsson
Examiner: Marcus Bendtsen
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

Robin Ellgren
© Tobias Löfgren
Students in the 5 year Information Technology program complete a semester long software development project during their sixth semester (third year). The project is completed in mid-sized groups, and the students implement a mobile application intended to be used in a multi-actor setting, currently a search and rescue scenario. In parallel they study several topics relevant to the technical and ethical considerations in the project. The project culminates by demonstrating a working product and a written report documenting the results of the practical development process including requirements elicitation. During the final stage of the semester, students form small groups and specialise in one topic, resulting in a bachelor thesis. The current report represents the results obtained during this specialization work. Hence, the thesis should be viewed as part of a larger body of work required to pass the semester, including the conditions and requirements for a bachelor thesis.
Abstract

High profile cyber attacks such as the one on DigiNotar in 2011, where a Certificate Authority (CA) was compromised, has shed light on the vulnerabilities of the internet. In order to make the internet safer in terms of exposing fraudulent certificates, Certificate Transparency (CT) was introduced. The main idea is to append all certificates to a publicly visible log, which anyone can monitor to check for suspicious activity. Although this is a great initiative for needing to rely less on CAs, the logs are still centralized and run by large companies. Therefore, in this thesis, in order to make the logs more available and scalable, we investigate the idea of a distributed client driven CT log via peer-to-peer (P2P) and WebRTC technology that runs in the background of the user’s browser. We show that such a system is indeed implementable, but with limited scalability. We also show that such a system would provide better availability while keeping the integrity of CT by implementing an append only feature, enforced by the Merkle Tree structure.
Acknowledgments

We would like to thank our supervisor, Niklas Carlsson for all the help and support we received. We would also like to thank Victor Nyberg, Daniel Holmberg, Jesper Holmström, Daniel Jonsson, Oscar Andell, Albin Andersson, Daniel Roos and Gustav Aaro for their valuable feedback.
# Contents

## Abstract iv

## Acknowledgments v

## Contents vi

## List of Figures viii

### 1 Introduction 1

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2  
1.2 Aim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2  
1.3 Research questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2  
1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2  
1.5 Delimitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

### 2 Theory 4

2.1 Public key distribution system . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4  
2.2 Certificates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4  
2.3 Certificate Transparency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5  
2.4 Distributed systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5  
2.5 Relevant data structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6  
2.6 WebRTC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

### 3 System design 11

3.1 Client side . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11  
3.2 Merkle Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
3.3 Peer discovery . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
3.4 System architecture summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

### 4 Implementation 15

4.1 Chrome extension . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15  
4.2 Centralized Merkle Tree server . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16  
4.3 Restrictions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

### 5 Validation and performance results 20

5.1 Chrome extension . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20  
5.2 Trackers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23  
5.3 Merkle Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

### 6 Related Work 26

### 7 Discussion 27

7.1 Validation and performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27  
7.2 System design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
List of Figures

2.1 The structure of a Merkle Tree with four certificate entries. .......................... 7
2.2 Audit proof on a Merkle Tree with four entries. ........................................ 7
2.3 Consistency proof on a Merkle Tree with four entries. ................................. 8
2.4 Simplified Chord protocol performing a lookup request. ............................... 9
2.5 A scenario for setting up a P2P connection with WebRTC. ............................. 10
3.1 The system interaction when downloading certificates from other peers. .......... 13
3.2 The system interaction when no peers with the certificate entry are found. .... 13
4.1 Our implemented Merkle Tree with 8 certificate entries, appended sequentially by peers. .......................................................... 16
5.1 The popup window produced by the extension in two different scenarios. ............ 21
5.2 The Chrome extension CPU usage with a varying amount of tabs ..................... 21
5.3 Thrown exception when limit on created RTCPeerConnection objects is reached. 22
5.4 Chrome resolving the createOffer promise in the extension’s background environment ................................................................. 23
5.5 Firefox not resolving the createOffer promise in the extension’s background environment ................................................................. 23
5.6 Torrent size vs download speed using two different machines and combining the results ................................................................. 24
In current times, it is possible to access virtually any service online and with the increased online activity follows the necessity of more reliable internet security. Being able to trust that a website is secure is becoming increasingly important and the level in which trust can be achieved is continuously increasing with the enhancements of the public key infrastructure (PKI) and secure transport layer communication (SSL/TLS) protocol\(^1\). However, this is still insufficient to ensure that Man In The Middle (MITM) attacks cannot occur. The introduction of certificates provided an extra layer of security, but problems still exist. The Certificate Authorities (CAs) that provide certificates and in whom websites and people trust can also be the victim of attacks.

In 2011 the Dutch CA DigiNotar was the victim of a hack, which resulted in the issuance of hundreds of fraudulent certificates for domains such as Google.com, Mozilla.com and many more\(^1\). Another example is Superfish on Lenovo computers\(^2\). Lenovo installed the local CA Superfish on its computers and Superfish used its position to inspect traffic and put up advertisement onto the users’ browsers. In both cases, the affected users’ browsers were exposed, enabling unwanted access to private information.

There needed to be a way to move the trust away from these companies and hence Certificate Transparency (CT) was introduced as a solution by Google in 2013\(^3\). The idea behind CT is for the CAs to append all of their issued certificates to a public log, where anyone can monitor the log and verify that a specific certificate was issued by the CA for a specific domain. While CT is a big step toward more secure browsing, there is still room for improvement, especially regarding the availability and scalability of CT. This thesis aims to provide a distributed technology for users to get the certificates in an efficient and safe way while browsing the web.

\(^1\)The 2011 Diginotar hack [http://www.slate.com/articles/technology/future_tense/2016/12/how_the_2011_hack_of_diginotar_changed_the_internet_s_infrastructure.html]\(\text{[1]}\)

\(^2\)A presentation discussing CT and the attacks against CAs [https://www.youtube.com/watch?v=tJFfDOQT46k]\(\text{[2]}\)

\(^3\)For more information about the founding of the CT project, see the project website: [http://www.certificate-transparency.org/what-is-ct]
1.1 Motivation

Peer-to-peer (P2P) technology introduced ways to effectively distribute large volumes of data between many users. Development within the field of cryptography has enabled users to not only distribute data, but also trust the integrity of the data to a higher degree. In the event involving DigiNotar, mentioned above, a CA was compromised and issued false certificates, possibly compromising the security of the whole internet. Google’s response to protect against similar events and hold CAs more accountable, i.e. the CT initiative, is quickly expanding to be a new web standard. The question still arises: if a CA’s integrity cannot be completely trusted, how come the CT logs’ integrity can? A combination of the P2P technology and the CT initiative, resulting in a distributed CT log, could potentially take advantage of benefits such as scalability and availability of service associated with the P2P technology and possible gains in data integrity. Such a system could still meet the main goals of the CT initiative - resulting in a more available and secure internet for everyone.

1.2 Aim

This thesis aims to further investigate the idea of CT and how it will impact the security of the web. More specifically, the idea of a distributed CT log is examined. The thesis focuses on implementation of such a system (as a proof of concept) and the analysis of the characteristics that such a system would have in the sense of the central key aspects: integrity, availability, performance and scalability. In order to accomplish this it is also important to understand why and how the certificate aspect is implemented in today’s public key infrastructure (PKI).

1.3 Research questions

With respect to the aim, we have identified the following research questions as the backbone of our project and the theme that permeates this thesis.

1. Can a distributed CT log be realized, and if so, how?
2. How will the integrity of the log be affected by such an implementation?
3. How will the implementation affect the availability of the log compared to the current implementation of CT?
4. Is the implementation scalable?

1.4 Contributions

This thesis has two main contributions that we want to highlight:

- Presenting the idea of a distributed CT log that could potentially replace or complement CT itself, by providing a proof of concept solution.
- Through P2P and WebRTC technology, enabling the usage of torrents within browser extensions.

1.5 Delimitations

Since this project aims to provide a proof of concept solution for a distributed CT log, the final implementation will more realistically provide the structure and ideas rather than the whole solution needed to realize a fully functioning distributed CT log that can be used on the web. Also, due to the scope and the time limit of the project, ideas that are too complex or time
consuming to finish within the given time frame will be simplified and/or overlooked. Additional discussions regarding simplifications are presented when individual simplifications are first introduced in the thesis, and their potential implications on the results are discussed in Chapter 5.
This chapter will discuss the different structures and technologies that will be used for the implementation later in this thesis. The exceptions are the sections 2.1 and 2.5.2.1, which are intended to provide insight about the different technical possibilities and solutions. In order to realize a distributed CT log, some choices had to be made and those will be presented later on in chapter 3 and 4.

2.1 Public key distribution system

In 1976 Whitfield Diffie and Martin E. Hellman proposed “approaches to transmitting keying information over public (i.e. insecure) channels without compromising the security of the system”[2], which came to be known as the public key distribution system.

In conventional encryption algorithms, the same key is used both for the encryption and decryption of messages. One of the most widely used conventional encryption algorithms is the NBS Data Encryption Standard (DES), where the key was considered safe in regards to brute forcing decryption of the message since creating a machine capable of it would cost twenty million dollars[3]. Today the DES is breakable by machines created for a lot less money. Even if we were to assume there existed an unbreakable conventional encryption algorithm, there are other issues that need to be addressed. As pointed out by R.Needham and M.Schroeder in 1978[4], the communication authorization then depends on the two communicating parties being the only parties with access to the key, which brings us to the problem of having to exchange the key beforehand, which can be inconvenient.

Enter the public key distribution system. This system instead proposes the use of public and private keys when sending messages between nodes. The main idea is that the public keys are free to distribute and do not have to be kept secret. Instead every node has a private key, which is not shared with anyone and can be used to decrypt messages that has been encrypted with its public key.

2.2 Certificates

In reality the implementation of the idea in section 2.1 is somewhat different than explained[5]. The basic idea with public and private keys does however still hold.
2.3 Certificate Transparency

is that a trusted node (CA) is needed to distribute said keys. A trusted CA issues a certificate
that holds the public key of the requested node, encrypted with the CA’s private key. A
node can request a certificate on itself which it later can use in communication with other
nodes. The receiving nodes can then use the CA’s public key (stored on their local machine)
to decrypt the certificate and validate the identity of the sending node. This implementation
significantly decreases the number of requests to the CA. If a receiving node does not hold
the CA’s public key locally, a certificate chaining process takes place where nodes ask the issuing
CA for its certificate. If the issuing CA is not trusted (i.e. its public key is not stored locally),
the node in turn checks the issuing CA’s issuing CA and so forth until one trusted issuer is
found. This process is further defined and explained in RFC 5280[6].

2.3 Certificate Transparency

Just as the name implies, CT makes certificates publicly visible (transparent), so that everyone
can inspect the certificates through auditors and monitors[7, 8]. The aim of CT is to “mitigate
the problem of misissued certificates by providing publicly auditable, append-only untrusted
logs of all issued certificates”[9]. All certificates are published in logs where CAs contribute
their issued certificates. The append-only property in the logs is achieved using the Merkle
Tree structure (see section 2.5.1)[10].

This structure ensures that whenever a certificate is appended to the log, the hash value
of the connected leaf, its connected node hash all the way up to the root hash is changed. This
means that anybody (since the logs are public) can notice whenever a certificate is appended
just by looking at the Merkle Tree Hash.

2.4 Distributed systems

There are various definitions of what constitutes a distributed system. Andrew S and Maarten
van Steen claim that a fitting definition is that “a distributed system is a collection of indepen-dent
computers that appear to its users as a single coherent system”[11]. One great benefit
of using a distributed system is availability: Simplifying the process of accessing remote re-sources
for the user and sharing resources in an efficient way. Another is scalability; being
able to remain effective when the number of users and resources increase significantly[12].
Since these features are both desirable and necessary, the distributed architecture will be used
for the implementation in this project later on.

2.4.1 The P2P architecture and BitTorrent

According to RFC 5694, a system is considered to be P2P if “the elements that form the system
share their resources in order to provide the service the system has been designed to provide.
The elements in the system both provide services to other elements and request services from
other elements”[13]. A P2P system can be considered P2P even if it is partly centralized by
using a centralized enrollment server, but the following functions need to be included in a
P2P system:

1. Nodes joining a P2P system need to obtain valid credentials to join the system. This is
done by using an enrollment function that handles node authentication and authorization.

2. In order to join a P2P system (become a peer), a node needs to establish a connection
with peers that are already part of the system. This is achieved through the peer discovery
function, which allows nodes to discover peers in the system in order to connect to
them.
BitTorrent is a P2P communication protocol for file sharing, with a distributed policy regarding peers interested in downloading and uploading specific content. BitTorrent’s architecture is based on the use of trackers, which is an entity responsible for peer discovery [14, 15, 16]. Despite the fact that BitTorrent is regarded by many as an efficient solution for file sharing, there are some issues that might arise with the use of centralized trackers.

### 2.4.1.1 BitTorrent with trackers

The first generation of BitTorrent used a single centralized tracker, which is a server responsible for establishing communication between peers. This does result in a single point of failure, meaning lower overall availability, as discussed by Neglia et al [17]. In the same paper, they are however able to prove that the use of multiple trackers improves the overall availability, since if one tracker goes down, peers can still connect through other trackers. The use of multiple trackers seems like a feasible solution for peer discovery, but other solutions need to be investigated before deciding which one to use in the implementation.

### 2.4.1.2 Trackerless BitTorrent

If failure occurs on the tracker entity, new peers are unable to discover other peers. A solution to this problem is to use trackerless BitTorrent [18] which leverages the Distributed Hash Table (DHT) structure (see section 2.5.2 for further explanation). The DHT stores the location of peers, so that peers, by using keys called magnet links (essentially a hash of desired data), can locate desired peers that they want content to download from or upload to [14]. When contacting other nodes, the messages are forwarded by the nodes in their respective routing table, which removes the single point of failure that the tracker solution entails [1]. However, using such an implementation increases the overall response latency [17]. This might not be a desirable feature when implementing a service over the web where only a small amount of data needs to be transferred.

### 2.5 Relevant data structures

#### 2.5.1 Merkle Tree

The Merkle Tree is a binary tree consisting of hashed leaves and nodes. The leaves are the hash of individual certificates entries that have been appended to the CT log. The nodes are the hashes of paired child nodes (or paired leaf nodes) and the root is constituted by the hash of every node in the tree and is known as the Merkle Tree hash. How the structure works and what advantages it brings is explained on the Certificate Transparency website [2]. The Merkle Tree structure is illustrated in Figure 2.1.

---

Certificate entries are appended at the bottom of the tree. They are then hashed and added as the leaf nodes. The leaf nodes are then combined into child nodes and so forth until the two remaining nodes at the second level of the tree are combined into the root (Merkle Tree hash). This structure enables the possibility to perform mathematical proofs to verify the integrity of the tree: Audit- and consistency proofs.

2.5.1.1 Audit proof

This proof is done in order to verify that a certain certificate has been appended to the log. To illustrate how the proof works, Figure 2.2 will be used.

In Figure 2.2 every node information that needs to be provided by the log and calculated by the one performing the proof is labeled accordingly. Every node necessary to perform the proof is colored: Orange nodes are calculated, green nodes are provided and blue nodes are both provided and calculated. The letters I, ii and iii show in what order the actions are performed. In this scenario, the audit proof is conducted by hashing the sought after certificate entry (i) and combining it with the parallel leaf hash connected to the same parent node (ii) and then again combining it with the parallel node hash (iii) in order to recreate the Merkle Tree hash. This proves that the certificate is in the tree, since there is no way of reproducing the same Merkle Tree hash without including the hash of the certificate entry in question.

2.5.1.2 Consistency proof

This proof is done in order to verify that certain certificates have not been removed from the logs or tampered with since the last time it was checked. This can be done if the old root hash
2.5. Relevant data structures

(i.e. the Merkle Tree hash from a previous state) is known. The proof will be illustrated using Figure 2.3 with the labels, colors and letters representing the same properties as in Figure 2.2.

![Figure 2.3: Consistency proof on a Merkle Tree with four entries.](image)

The tree is consistent if it is possible to verify that the old root is a subset of the new root. Using Figure 2.3 as the example where the old Merkle Tree hash is known (i.e. an old state of the tree is known). The tree would then need to provide the newly appended certificate entries. By hashing these entries into leaf hashes (i) and by combining the leaf hashes into a node hash (ii), it is now possible to combine the calculated node hash of the newly appended certificate entries with the old root hash (iii) to form the new Merkle Tree hash. If this is possible to accomplish, it is proved that the old Merkle Tree hash is in fact a subset of the new Merkle Tree Hash and that the tree therefore is consistent.

2.5.2 Distributed Hash Tables (DHT)

A fundamental problem when dealing with P2P-architected applications is the problem of knowing which peer holds what data. One way of dealing with this problem is using a Distributed Hash Table (DHT)\[19\]. A DHT is in its essence just like an ordinary hash table that maps a hashed key to a corresponding value. Translated to P2P terminology, a DHT will map a key (for example a torrent) onto one or more nodes (peers) holding the particular data object. An implementation of a DHT would need to also rapidly adapt to nodes (peers) joining and leaving the network and distribute the content accordingly\[19\].

2.5.2.1 Chord

One protocol that uses DHT is the Chord protocol. The Chord protocol was presented by Ion Stoica et al. in 2003\[20\]. The protocol orders the peers in a Chord ring by assigning every peer an m-bit long identifier by hashing their IP-address with the SHA-1 hashing algorithm. The same procedure is done with the keys, enabling them to also be placed into the Chord ring. Key \( k \) is then assigned to the first node with an identifier that is equal to or greater than the key itself. In Figure 2.4 we can for example see that K54 is assigned to N56. The node that holds the key is called the successor of the key. The closest existing node with an identifier greater than another node is also called the successor of that node, for example N14 is the successor of N8 in Figure 2.4.
In Figure 2.4(a), pseudocode for looking up the successor of a particular key is presented. This approach makes the nodes only hold a very small amount of data, namely their current successor in the Chord ring. That does obviously increase the lookup time linearly with the amount of nodes, i.e. with a time complexity of \( O(N) \) where \( N \) is the number of nodes. One way to decrease lookup time (which is what the Chord protocol has done) would be to let each node hold a so called *finger table*, effectively a small routing table to nearby nodes. The finger table should then be updated continuously and thus effectively reduce lookup cost to a time complexity of \( O(\log(N)) \). Such a time complexity would provide scalability if it was to be used in our implementation\[20\].

The approach described in Figure 2.4 is called *consistent hashing* and has the advantage of being easily implemented. Another major advantage is the fact that nodes can join and exit the network without the need for refactoring the whole keyset. When a node \( n \) joins the network, it simply notifies its successor and predecessor of its existence and takes over responsibility of a certain keyset. Similarly, when node \( n \) leaves the network, the responsibility of its keyset is simply moved to its successor.

### 2.6 WebRTC

Web Real-Time Communication (WebRTC) is an open standardized technology in effort to enable direct communication between browsers without the need for servers\[3\]. In classic web architecture, the client-server paradigm is mostly used. WebRTC instead uses the client-server model for setup and then introduces the P2P paradigm between browsers. This is illustrated in Figure 2.5 as presented by Loreto et al\[21\].

\[3\]The WebRTC official homepage can be found at: [https://webrtc.org](https://webrtc.org)
In order to create an RTCPeerConnection\(^4\) establishment, several steps need to be executed. In Figure 2.5, this is labeled as setup. The steps are as follows: \(^2\)

- Requesting the establishment of an RTCPeerConnection to a peer
- The web server routes the request to the peer
- The peer approves or refuses the RTCPeerConnection
- Exchange of necessary parameters (this includes address information of the peers)

When the connection has been established, the peers are free to communicate over P2P without the need for servers. As pointed out by Jennings et al, a key feature of the WebRTC architecture is that it allows multiple established connections per peer\(^2\). Therefore, WebRTC can be used in a wide variety of contexts, such as P2P chat rooms and torrent distribution.

---

\(^4\)For more information about the RTCPeerConnection object, please see the developer API: [https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection](https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection)
3 System design

As described in section 1.3, one of our main goals with this thesis is to investigate whether it is possible to implement a distributed CT log. Our goal is to implement a proof of concept system with a client driven approach which, with further development, users can use as a complementary alternative to CT itself.

A CT log has three important qualities\(^1\) and in order to create a complementary alternative it is absolutely necessary that our system achieves the same three qualities:

- **Append-only** – certificates can only be added to a log; certificates can’t be deleted, modified, or retroactively inserted into a log.
- **Cryptographically assured** – logs use a special cryptographic mechanism known as Merkle Tree Hashes to prevent tampering or misbehavior.
- **Publicly auditable** – anyone can query a log and verify that it is well behaved, or verify that an SSL certificate has been legitimately appended to the log.

In the remainder of this chapter we will evaluate different design approaches in order to motivate our choices before implementing them. The choices are mainly based on distributed aspects and the desired CT qualities.

3.1 Client side

As described in section 2.3, the peers are expected to be given proof of log entries. We wanted to include the trait that the log should work seamlessly while the user is browsing the web and therefore we decided that the client side of this system should be implemented as a browser extension. The most widely used browser is Google Chrome with approximately 57%\(^2\) of the users, making the decision towards building a so called *Chrome extension* fairly easy.

\(^1\)The qualities are quoted from the Certificate Transparency official website: [http://www.certificate-transparency.org/how-ct-works](http://www.certificate-transparency.org/how-ct-works)

\(^2\)This information is approximate, but still gives an indication. Please see the StatCounter website for more browser statistics: [http://gs.statcounter.com/browser-market-share](http://gs.statcounter.com/browser-market-share)
3.2 Merkle Tree

Ideally, the distributed CT log would implement a distributed Merkle Tree over a DHT. The basic idea is for peers to represent nodes in a Merkle Tree, thus dividing the data among the peers in the tree. A Merkle Tree for this purpose would naturally be very sensitive to structural changes in data locations (i.e., nodes leaving and joining the swarm) because we need to identify certain tree nodes to be able to reproduce proofs. A possible solution could be to use a route distribution approach which effectively lets every tree node know about its children and every leaf node know its path to the root, enabling the peers to recreate a Merkle Tree, as described in further detail by Tamassia et al[23]. The centralized approach of the Merkle Tree is how CT is built today and with regards to the mentioned sensitivity of a distributed Merkle Tree, we decided to keep the status-quo on that part.

3.3 Peer discovery

Peers discovering peers who hold desirable data is the heart of our implementation. Also the reverse, announcing that a peer holds desirable data is a core feature. Essentially there are two options to achieve this: trackers or DHT.

The torrents (certificate log entries) will only be JSON data and only a few KB of size, as shown in section 5.2. It is then reasonable to expect that the download should happen almost instantly. Given this point of view, we decided to go with the tracker approach instead of DHT. Consider the following example of why:

Let the system have $N$ users and let $N = 10^7$. This amount is not unreasonable (more likely an underestimate), if we assume the system will be built in with every Chrome browser, and not just downloaded from the Chrome web store like a regular extension. Let $RTT$ be the average Round trip time in the network. Then, the lookup latency will be $O(1) \cdot RTT \approx RTT$ for the tracker approach and $O(\log(N)) \cdot RTT \approx 23 \cdot RTT$ for DHT with a finger table approach. With an RTT of just 200ms the difference is as big as 4.4s between the approaches. Given that we expect the torrents to be downloaded instantly, especially regarding their size, the DHT approach is not a good choice.

3.4 System architecture summary

The result of using all of the features described above is a system with the following features:

- A Chrome extension that serves as a proof of concept for the client side.
- Two tracker servers (that could be extended into any number of trackers) for peer discovery instead of a DHT. This provides a better availability compared to the current implementation of CT since there now is no single point of failure.
- A centralized Merkle Tree server (as opposed to a distributed Merkle Tree), that acts as a backup way to get certificate entries if peer discovery fails or if the requested torrent is not seeded. When used, the centralized Merkle Tree also provides audit- and consistency proofs.

The architecture of the system when the extension is used is described in Figures 3.1 and 3.2. The first one illustrates how peers get certificates from other peers using the extension and the second one illustrates how peers get the certificates from the centralized Merkle Tree (Merkle Tree server) when no other peer is found. The arrows in the figures show in which direction data is sent and the numbers on the arrows show in what order the actions are executed.
3.4. System architecture summary

In Figure 3.1, the numbers on the arrows represent the following system interaction:

1. The user browses Chrome with the extension installed.
2. When visiting a website, certificate information is retrieved.
3. The user fetches the certificate information and creates a certificate entry as if it existed in the log. Said log entry is hashed and the user then knows which torrent to download.
4. The user requests peer information about the torrent from the trackers.
5. If a peer with the torrent is found, the tracker returns a connection to said peer.
6. The user downloads the torrent from the peer.
7. The user announces the torrent to the trackers so that other peers can download it.

In Figure 3.2, the numbers on the arrows represent the following system interaction, with the first four steps being the same as in Figure 3.1:

Figure 3.1: The system interaction when downloading certificates from other peers.

Figure 3.2: The system interaction when no peers with the certificate entry are found.
5. The tracker timeouts, and thus the peer verifies that the entry was in the log by requesting it from the centralized Merkle Tree server.

6. The server responds with a result and the user performs audit- and consistency proofs to verify the integrity of the server.

7. The user announces the certificate entry (torrent) to the trackers so that other peers can download it.
4 Implementation

This chapter show the implementation of our system design presented in chapter 3. We will thoroughly present our technical solutions and problems, as well as the restrictions that need to be addressed.

4.1 Chrome extension

Firstly we needed the browsers to be able to communicate with each other via a P2P network. BitTorrent is currently one of the most widely used protocols for this type of communication, although it is not built for browser to browser communication. We believe that the foundation of the BitTorrent protocol, with files being hashed into torrents and currently seeding peers being available with trackers, can still be used for this purpose though. The open source project WebTorrent provides just this, namely a browser based torrent client built on the BitTorrent protocol. The major difference with the BitTorrent protocol is that instead of directly using TCP (or any other transport protocol) it uses WebRTC to enable the communication between the browsers.

The WebTorrent project is not just built on the WebRTC project, but also a significant amount of other open source projects, although all of them with a common denominator: They all run on top of the NodeJS core. NodeJS is in its essence intended to run on a server to provide content to its clients. This presented us with a problem, since a Chrome extension is not a server environment. We decided to take advantage of the Browserify project to embed the NodeJS core as well as all other required packages into a single file that we could dynamically load into the extension.

Given the architecture in chapter 3 and weighing in the possibility that the centralized Merkle Tree server might not respond, the following 3 scenarios could potentially happen when using the extension:

- A log entry was received from peers.
- A log entry could not be received from peers, but was instead received from the Merkle Tree server (how CT works today).

---

1 For more information about the WebTorrent project, see their homepage: https://webtorrent.io
2 For more information about the NodeJS project, see their homepage: https://nodejs.org
3 For more information about the Browserify project, see their homepage: http://browserify.org/
4.2 Centralized Merkle Tree server

The implementation for the Merkle Tree was based on an open source project in Python.

The Merkle Tree is structured as a file directory with the certificate entries in text files, named numerically based on when they were appended, as seen in Figure 4.1. It is a simplification of the Merkle Tree presented in the theory chapter, since it is a tree with only two levels, as opposed to a binary tree. The simplification of the implementation is justified in section 5.3. This implementation does however still provide the possibility to perform the same proofs as in a Merkle Tree, namely audit- and consistency proofs, but the proofs are conducted differently from the ones described in Chapter 2. How the proofs are performed for our implementation will be described further below.

\[
R = h(h(1)||h(2)||...||h(8))
\]

4.2.1 New audit proof

Recall that audit proof is the way for a peer to know for sure if a certain certificate is in the log. This is done whenever a user sends a request to the server. The server then provides the necessary tools for the user to verify that the certificate was appended to the log. This is accomplished with the server hashing the requested log entry and checking where in the log the certificate with that hash is located. In this scenario, two possible cases can occur:

1. The certificate was not in the log

If this is the case, the server appends the certificate as the last entry in the log and responds with the concatenation of the individual leaf hashes of every previous entry, as well as the new root hash which acts as a checksum for the user. Using Figure 4.1 as an example, if a certificate entry

Merkle Tree implementation in Python: [https://github.com/sangeeths/merkle-tree](https://github.com/sangeeths/merkle-tree)
new certificate entry would be appended to the log, the certificate entry would be appended as the 9th element and the server would respond with the concatenated hash string:

\[ h(1)||h(2)||...||h(8) \]

as well as the new root hash:

\[ R = h(h(1)||h(2)||...||h(9)) \]

With this information, the user can hash their own certificate (which in this case corresponds to \( h(9) \)) and then concatenate it to the hash string provided by the server and then hash again, which yields \( h(h(1)||...||h(9)) \). If the calculated root hash is the same as the root hash provided by the server, the user can guarantee that the certificate was not in the log before, but now is. This would in turn mean that the server indeed can be trusted in this case.

2. **The certificate was in the log**

If the certificate is already in the log, the same logic is applied. The server locates the certificate entry and responds to the user with the root hash, the concatenation of the hash of the entries appended prior to the user’s entry as well as the concatenation of the entries appended after. With this information, the user can concatenate the hashes with their own hashed entry and check if the hash of that concatenation gives the same root hash as provided by the server. If true, then the requested certificate was de facto in the log. Still using Figure 4.1 as an example, if the user was to ask the server if entry 5 had been appended to the log, the response from the server would be:

Certificates appended prior to requested certificate:

\[ h(1)||h(2)||h(3)||h(4) \]

Certificates appended after the requested certificate:

\[ h(6)||h(7)||h(8) \]

By hashing their own certificate entry, which gives \( h(5) \), the user can now concatenate the strings and calculate the root hash:

\[ R = h(1)||...||h(5)||...||h(8) \]

If the calculated root hash is the same as the one provided by the server, the user is guaranteed that the certificate was already appended to the log.

4.2.2 **New consistency proof**

Recall that a consistency proof checks whether or not a new state of a Merkle Tree is consistent with an old state. This means that given an old root hash, a user can verify if a new root hash contains all the nodes from the old root hash and no certificate has been removed or manipulated (i.e. the old root hash is a subset of the new root hash). The users saves the root hash locally from the last interaction with the server to use it the next time they send a request. This is done in order for the user to continuously check the integrity of the log. The server takes the root hash from a user and shows the user that it can be reproduced from the Merkle Tree.

The fact that the root hash is calculated by hashing every certificate in order can be used. The server tries to reproduce the user’s root hash by hashing the first certificate \( h(1) \), then the first and the second \( h(1)||h(2) \) and so on. By doing this, if the server manages to reproduce the user’s root hash, it simply needs to provide the user with the following:
4.3 Restrictions

4.3.1 RTCPeerConnections

We found during implementation that there was a limit on the amount of created RTCPeerConnection objects that can exist. It is essential that we find out what this limit is because otherwise we will not accurately discuss the scalability of the application. We performed an empirical study trying to reach the limit and recognized that we can create RTCPeerConnection objects until the limit is reached and the Chrome console outputs an error. The result can be found in section 5.1.2.

4.3.2 Certificate data

Chrome does not provide an API for accessing the certificate data. This means that there is no way for our extension to get information about the certificate of the site that the client is visiting. Without this information, it is impossible to build a fully functioning CT log in Chrome. We do however only strive toward completing a proof of concept solution and will as an effect of this just use mockup data instead of certificates. Why Google has decided to not offer this API is because the mapping between sockets and requests are problematic. The issue has been marked as a “wont-fix” and there is currently no indication of this changing anytime soon.\(^5\) Noteworthy is that there are some experiments for this issue, but the solution is only available for ChromeOS.\(^6\) Since the certificate data is available in Chrome via the “Secure website” tool we recognize that the proof of concept solution using mockup data is feasible since an implementer such as Google could implement the solution with actual data. Other solutions to this could have been to call CAs’ APIs (if they existed) or creating an own external API, although that would require keeping the API data updated, which would be a problem since there is no efficient way of doing it.

\(^5\) Read more about the issue in the Chromium bugs portal: [https://bugs.chromium.org/p/chromium/issues/detail?id=107793#c20]

\(^6\) More information about the chrome.certProvider API: [https://developer.chrome.com/extensions/certificateProvider]
4.3.3 Distributed Merkle Tree

Ideally we would like the implementation to consist of a distributed Merkle Tree. The idea was for the centralized aspect to be removed by creating a Merkle Tree of the peers using the extension, but it proved too difficult to implement without apparent errors, some of them mentioned in section 3.2. The implementation is instead referenced as an idea of future work in section 8.1.
5 Validation and performance results

This chapter strives to validate that our implementation meets the qualities that a CT log should have, as referenced in chapter [3]. Further, it validates that the implementation uses a distributed approach, with some alternations where deemed necessary. Lastly, we measure the performance of said implementation.

5.1 Chrome extension

We were able to implement a system where all the required qualities are met. The approach is client driven and one can successfully download and seed torrents directly from the extension, with no additional software necessary. Our implementation uses two trackers wss://tracker.btorrent.xyz and wss://tracker.openwebtorrent.com, which removes the single point of failure when downloading from other peers, compared to the current implementation of CT and thereby increasing the overall availability. Using two trackers is completely arbitrary and the number of trackers is customizable. The trackers are open source and anyone can host their own using the BitTorrent-tracker package[1]. Letting one user enter a website and just briefly after, with another user, visit the same website produces the expected behaviour that the first user receives the message “server” and the second user receives the message “peer” in the provided by section in the extension, as seen in Figure 5.1. In the Figure two different scenarios are shown for the produced pop up windows: The first one when downloading the certificate from the server and the second one when downloading it from another peer. The implementation is set to seed tabs through an entire session (i.e. until Chrome exits).

[1]The BitTorrent-tracker package is available on GitHub, see: https://github.com/webtorrent/bittorrent-tracker
5.1 Chrome extension

![Image](https://via.placeholder.com/150)

(a) Validated by server  
(b) Validated by peer

Figure 5.1: The popup window produced by the extension in two different scenarios.

5.1.1 CPU usage

An important aspect of distributed systems, as mentioned in 2.4, is the scalability; being able to maintain great performance regardless of the amount of data in the system. With that said, if peers using the extension maintain a large amount of tabs up at the same time, it is important that the CPU usage is not overwhelmingly high, since it might affect the performance. This can be evaluated by monitoring the CPU usage in Chrome while varying the amount of tabs the user has active.

![Graph showing CPU usage](https://via.placeholder.com/150)

Figure 5.2: The Chrome extension CPU usage with a varying amount of tabs

Figure 5.2 shows the Chrome CPU usage for a user browsing the web with the extension installed and with a varying amount of active tabs. The experiments were conducted using multiple machines because of a desire to avoid the bias introduced with only one data source. The machines were both laptops, while machine 1 runs on Mac OS X, machine 2 runs on Windows 10 x64. The interval estimate of the measurements in the figure is illustrated with a 95% confidence interval.

The general behaviour is that the CPU usage steadily increases, with varying fluctuation and the cap is reached at around 36 tabs. It should be noted that the data presented in 5.2 is not continuous, but in bursts. In order to keep the data on the tracker side updated, a peer continuously needs to announce to the tracker that it still holds the data. This is done because
we do not want to waste time trying to establish a connection to an offline peer, because time
is an important aspect in our system, as argued in section 3.3. Nevertheless, this means that
there is a limit on how many tabs a user can have active at the same time and in turn, how
scalable the system is.

5.1.2 RTCPeerConnections

There is a limit on the number of RTCPeerConnections that be can be alive simultaneously.
As described on the Chromium issues page and verified with a small script, seen in the code
below (that creates and then closes RTCPeerConnection objects), we find that the limit is
hard coded (i.e. not system dependable) into chrome as 500, even though the connections are
closed and unreferenced.

```javascript
<input type="number" value="1000">
<button type="Button">Go!</button>
<div>0</div>

document.querySelector('button').addEventListener('click', () => {
  const limit = document.querySelector('input').value;
  const result = document.querySelector('div');
  let p = Promise.resolve();
  let counter = 0;
  for (let i = 0; i < limit; ++i) {
    p = p.then(() => {
      return new Promise((resolve) => {
        (new RTCPeerConnection()).close();
        ++counter;
        result.textContent = counter;
        setTimeout(resolve);
      });
    });
  }
});
```

A script that creates and closes RTCPeerConnection objects. The number of created connec-
tion objects continues to rise toward the limit until an exception is thrown:

![Uncaught DOMException: Failed to construct 'RTCPeerConnection': Cannot create so many PeerConnections](image)

Figure 5.3: Thrown exception when limit on created RTCPeerConnection objects is reached.

The reason for this event is that even though the connection objects are unreferenced (set
to null), they are not truly removed (i.e. no decrementation on the amount of currently cre-
ated connection objects) until the Chrome garbage collector comes into play. The connection
objects do however not take up that much memory and therefore the garbage collector is not
called before the limit is reached, which results in the exception even if we remove unused
connections. This restriction on our implementation is further discussed in chapter 7. Run-
nning the same script with Mozilla Firefox shows that the problem is solvable because Firefox
does not suffer from the same limitations.

---

2The whole discussion can be found at: [https://bugs.chromium.org/p/chromium/issues/detail?id=825576](https://bugs.chromium.org/p/chromium/issues/detail?id=825576)

A runnable version of the small script can also be found at JSFiddle: [https://jsfiddle.net/0tp6tzhu/](https://jsfiddle.net/0tp6tzhu/)
5.2. Trackers

5.1.3 Firefox extension

this hard coded limit put a lot of constraints on our extension, especially regarding its scalability, we decided to try and port the extension into Mozilla Firefox. Porting a Chrome extension into Firefox was an easy process since the extensions' APIs are directly compatible. There are however a few exceptions where the Chrome namespace is intentionally unsupported but we did not encounter any issues with that. The implementation appeared to work in Firefox, but a few tests showed that peers were unable to communicate with each other. Some research showed that Firefox implements another policy than Chrome regarding permissions for the extension. Firefox does not let extensions use WebRTC in the background (i.e for all tabs simultaneously), just one at a time. This means that only the user’s currently active Firefox tab is seeded, as opposed to all tabs in a Chrome extension. This behaviour is confirmed by multiple issues on the Firefox bug reporting platform Bugzilla, but it is an intended privacy feature rather than a bug.

The behaviour can be reproduced by setting up a WebRTC connection in the background of the extension. This is done by creating a new RTCPeerConnection object and then attempting to generate a new offer, inside the extension’s background environment.

```javascript
new RTCPeerConnection().createOffer(console.log, console.error)
```

\[ Promise \{<resolved>: undefined}\]

```javascript
new RTCPeerConnection().createOffer(console.log, console.error)
```

\[ Promise \{<state>: "pending"}\]

Figure 5.4: Chrome resolving the createOffer promise in the extension’s background environment.

Figure 5.5: Firefox not resolving the createOffer promise in the extension’s background environment.

As shown in Figure 5.4 and 5.5 Firefox does not resolve the promise of generating an offer on the newly created RTCPeerConnection. Without that promise, which gathers necessary information from the browser, it is not possible to establish a connection to other peers.

5.2. Trackers

In section 3.3 we discussed the choice between trackers and a DHT. We argued that using trackers for peer discovery was the most suitable option for us, due to the advantage they bring regarding the time it would take to transfer a small torrent from one peer to another. The tracker solution can be evaluated by measuring the download time for torrent files with varying sizes (the times include peer discovery).

How to port an extension from Chrome into Firefox: https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Porting_a_Google_Chrome_extension

Issue 1: https://bugzilla.mozilla.org/show_bug.cgi?id=1398083

Issue 2: https://bugzilla.mozilla.org/show_bug.cgi?id=1278100
Figure 5.6: Torrent size vs download speed using two different machines and combining the results

Figure 5.6 shows the download time for torrent files with varying sizes and for different machines. In the same way as done in Figure 5.2, multiple machines were used in order to avoid the bias introduced with only one data source. Machine 1 runs on Mac OS X, machine 2 runs on Windows 10 x64 and Figure 5.6c shows the combination of both machines. The X-axes are logarithmically scaled while the Y-axes are linear. This is done for diversification of data points. Note that without scaling, the download time increases proportionately with the file size, and hence a linear trendline was used to highlight that behaviour. In all three Figures, the interval estimate of the measurements is illustrated with a 95% confidence interval.

As for the results, the download time ranges between 200 to 400 ms with relatively small torrent files, between 1 kB and 100 kB, regardless of what machine was used. What happens is that the tracker establishes a connection between two peers, whom handshake and then proceed to send the data, with almost no additional time with such a small amount of data. With larger files, over 100 kB, the download time increases rapidly. A torrent file with the size of 500 kB is downloaded in close to 800 ms. This is not surprising, since without logarithmic scaling, the download time, as mentioned, increases proportionately with the file size. This means that if the system was scaled with larger torrent files, the tracker solution would still be feasible. Despite the fact that the confidence interval vary for different machines, the
overall behaviour is similar for all Figures, effectively meaning that the tracker solution is not machine-dependent.

The data labeled with a bubble marked in Figure 5.6 shows the time it takes to download a file the size of an X.509 certificate log entry from one of Google’s CT logs, Rocketeer. The time it takes for the trackers to establish a connection between peers, without sending data, is close to 200 ms. As expected, the download time for a torrent file of a certificate entry is very close to that, validating that trackers indeed provide a fast solution for our implementation.

5.3 Merkle Tree

The distributed aspect of the Merkle Tree aspect was not achieved (explanation of why is found in section 4.3.3) and instead a centralized Merkle Tree was used by every user in the distributed system. This still made it possible for clients using the extension to verify the integrity of certificate entries provided by the server without removing the distributed aspect of the extension itself.

Instead of using the tree aspect to store the certificate entries and to provide proofs for the user, this was accomplished by appending the certificate entries as a leaf hashes with one root, which still provides the audit proof and consistency proof in order for the user to verify the integrity of the log. We used this implementation because it was simpler to implement and at the same time, from a research perspective it does not affect the result.

Performance-wise, our implementation of the Merkle Tree provides linear time complexity, $O(N)$ where $N$ is the number of log entries, as opposed to the time complexity $O(\log N)$ that the normal Merkle Tree provides. The linear time complexity in our implementation applies to both the audit proof and consistency proof, since they both try to recreate a given hash by concatenating the log entries sequentially.
6 Related Work

This thesis investigates two key areas: distributed systems and CT logs. In order to understand these areas fully and utilize their advantages, we considered some projects that provided great insight and work in these areas. Due to the alternations of the initial distributed CT log idea, brought on by the implementation, these areas were emphasized in different magnitudes.

**Distributed systems.** Distributed systems proved during the implementation to be the aspect in which we had the most leeway to deploy different structures as we saw fit. For example, Webtorrent, that was used as the framework for the extension, provided built-in possibilities for both the use of a DHT and trackers for peer discovery. The biggest decision regarding the distributed aspect was which peer discovery solution to use. Neglia et al. [17] provided great comparisons between the solutions and came to the conclusion that the use of a DHT or multiple trackers improved the availability compared to the single tracker solution, but that the DHT solution induced high response latency. This finding made us decide to use the multiple tracker solution for peer discovery, since it has the better qualities in a CT log context.

**CT logs.** The CT log aspect was not as much emphasized as the distributed aspect, since it was not possible to access certificates through Google’s API. However, we were able to implement the Merkle Tree structure that the CT logs have. Gustafsson et al. [7] provided a comprehensive overview of the CT landscape and its characteristics, which helped us understand the idea behind CT and to come up with ideas of how to implement a CT log.

**Distributed CT logs.** One of our initial ideas was to combine these aspects and create the Merkle tree aspect of a CT log over a DHT, as mentioned in section 3.2. When DHTs were developed, various distributed trees were presented [24, 25]. Tamassia et al. [23] evaluated these trees and found them incompatible with a distributed Merkle Tree due to the sensitivity of the cryptographic function when nodes leave and join the swarm. Tamassia et al. instead propose the usage of a route distribution approach, but does not provide any clear insights on the topic of audit- and consistency proofs. An implementation addressing these questions would be of great benefit to the idea of a distributed CT log and is therefore referenced as possible future work in section 8.1.
This chapter discusses the results found by validating and performance testing the implementation. We also evaluate what we could have done differently regarding some of our design choices. Lastly, we talk about possible ethical and societal implications and our work in a wider context.

7.1 Validation and performance

In section 5.1.2 we showed with a small script that the number of created connections in Chrome is limited to a hard coded number of 500. The fact that the counter is non-decrementable means that our system is bound to eventually crash. Despite the fact that Firefox solves this issue, not being able to seed more than one tab at a time (i.e. not being able create and respond to offers in the background of the extension), is according to us such a huge drawback to the scalability of the system that we deemed it unnecessary to investigate Firefox further.

The limit strongly relates to the question whether it is applicable for many users to seed the same log entries. One could imagine that popular websites such as for example Facebook.com, Google.com and Youtube.com would quickly get millions of seeders, which would result in a major overhead. This would probably not be a problem for the peers, especially if the peer connection limit was decrementable or more flexible (like in Firefox), but could become a problem on the tracker side since they would have to handle continuously increasing amount of overhead traffic.

Furthermore, with regards to the system performance, we presented results in section 5.1.1 about how the number of seeded tabs correspond to the CPU usage. We see that when we approach approximately 30-40 tabs the CPU usage starts to hit a very high percentage. This means that any real application attempting this implementation would have to consider some sort of least recently used queue where the least recently used log entry stops being seeded, in order to improve the performance.

7.2 System design

The choices of the system structure changed from the initial idea, due to the limitations that were discovered during the implementation.
7.3 Implementation

For instance, the choice to implement a Chrome extension resulted in less possibilities to realize the CT part of the project because of the restrictions in the Chrome API regarding certificate information. As a consequence, instead of using certificates, we used mockup data and the distributed aspect of the project was more emphasized. Thus most of the effort was put into implementing P2P functionality over the web. Despite the lack of actual certificate information, we were still able to provide the Merkle Tree aspect of CT, more specifically the audit- and consistency proofs, and in turn the data integrity methods that CT entails.

Moreover, we set out to make the distributed system trackerless (i.e. using a DHT). Several implications made trackers a much more suitable options. The biggest reason to use trackers instead of DHT was that the average response delay of a DHT was too high in comparison to the trackers’, when receiving certificate entries from other peers. When it comes to larger torrents such as movies this delay is acceptable, but when it comes to browsing the web, we felt that torrents containing certificate entries need to be provided faster for a better user experience.

7.3 Implementation

There were a couple of choices on which open source projects to use in order to implement the system. Chrome extensions are only supported on web based languages (HTML, CSS and Javascript) and because of this, many interesting projects regarding either P2P or DHT were discarded because they were not supported. Luckily, we found a way to create a P2P system over the web with the WebTorrent API. The main advantage of this implementation was the simplicity to apply it to a web extension and the option to choose different structures to fit our project as we saw fit. For example, support was provided for both the use of trackers and DHT. This gave us more leeway when implementing.

Regarding the idea of implementing a distributed Merkle Tree, we decided to use a centralized approach instead, mostly because of the structural sensitivity, as discussed in 3.2 and the complexity of designing a fully distributed tree structure divided between many peers. There were however some interesting ideas we considered using, such as the one presented by Tamassia et al. and briefly introduced in section 3.2.

7.4 The work in a wider context

While the idea of a truly decentralized internet seems closer than ever with the rise of for example Bitcoin, big cloud service players such as Amazon Cloud Service and Microsoft Azure are gaining market shares every day as seen by multiple recent news articles. This means that even though the internet is designed as a distributed system, we can see tendencies that it is moving closer to a centralized unit. This is also applicable to the idea of Certificate Transparency, where a few major players control most of the logs, even though it is open for anyone to start their own.

Our work questions this centralizing development by suggesting the alternative approach of a distributed CT log. It should be noted that our system is not ready for any large scaling and is merely to be used as a proof of concept that CT log entries can be shared between clients using WebRTC technology.

In order to further develop our distributed approach to a large scaled application, it would have to be implemented directly into all major browsers. This does however seem highly unlikely, mainly because the tracker solution is not really scalable to the levels required without using improvements such as load distribution and that the DHT approach is too slow.

When it comes to societal and ethic aspects of this project, we try to address them by refraining from storing sensitive information about users. Also, we acknowledge the possi-

---

1 One example from Financial Times: [https://www.ft.com/content/6abc4574-4973-11e8-8ee8-cae73aab7cc2](https://www.ft.com/content/6abc4574-4973-11e8-8ee8-cae73aab7cc2)

bility that illegal file sharing might be more accessible with this technology in the hands of the wrong person.
8 Conclusion

We were able to implement a distributed CT log with a few alternations. The implementation is better than or equal to (in the worst case) the current implementation of CT when considering the availability that the extension provides. Regarding the integrity of the log, it is the same as the current solution, but the question of peers seeding log entries that are not actually in the log is an open issue but can be solved by either confirming the data and monitoring the state of the centralized Merkle Tree server or by implementing a distributed Merkle Tree.

By taking advantage of WebRTC technology and the BitTorrent protocol we were able to bring torrents into a Chrome extension, which makes the system very easy to use while browsing the web. When designing the system we decided to use Chrome. Firefox seemed promising as an alternative, since it did not suffer from the same scalability issues, but porting the extension from Chrome to Firefox brought on other scalability issues. Since we saw no point in trading one big scalability issue for another, we decided to keep the platform used for this thesis and instead investigate other areas. Chrome put restrictions on our system, but as a proof of concept solution it suffices to provide the key features of a distributed CT log.

When it comes to peer discovery, we arrived at the conclusion that using trackers is a more feasible solution. The alternative would have been to use a DHT instead, which is more scalable, but with respect to providing quality of service, this option was insufficient and was therefore discarded. Regarding performance, the system uses a fairly high amount of CPU, but it might be possible to control it by limiting the amount of maximum seeded log entries one client can have at any given moment or to load balance the trackers.

As for our aim we consider it to be reached, although we acknowledge that there is still plenty to investigate with the idea of a truly distributed CT log.

8.1 Future work

This project presents a proof of concept solution for a distributed CT log, but in order to develop the system into a more functional application that can be used in a larger environment, there are a couple of improvements that needs to be done. These improvements mainly consider the distributed aspects. We have a lot of ideas, but there are two main ideas listed below on different implementations that can be added to enhance the system significantly.
8.1. Future work

8.1.1 Distributed Merkle Tree aspect

Our implementation uses a centralized Merkle Tree. If one instead could place the responsibility of the log integrity onto a distributed system, the trusted entity could be removed, which would bring the idea of a distributed CT log one step closer to reality. Other major communities such as the torrent community would also heavily benefit from this. Important questions such as how to handle rouge nodes, concurrency issues and how to conduct audit and consistency proofs would need to be addressed. As an immediate effect of those questions, one would also have to think about how to guarantee that no data is lost even if peers ungracefully leave the swarm.

8.1.2 Popularity based load balancing on trackers

In the current implementation, each of the trackers possess the same data (i.e. connections to resources). A problem that arises with this solution is that one certificate entry is seeded in too excessive amounts. The system is structured this way because of the need for redundancy in the system, but the implementation encounters issues with scalability due to too much redundancy. One approach would be to limit the amount of times a log entry can be seeded based on the popularity of a specific entry, resulting in popular entries being found frequently among the trackers while unpopular entries would be found less frequently. This was done in a similar fashion by Neglia et al.[17]. By using information on how popular certain websites (and in turn certain certificates) are, the trackers can distribute resources accordingly. Given such an implementation, the system as a whole would possibly be more scalable.

Another problem with the BitTorrent implementation is that since a peer can only be associated with one tracker per file, if the peers associate with different trackers, many small swarms (as opposed to a large swarm) are created for unpopular torrents. One solution would be to manage the swarms between the trackers dynamically in a similar fashion as Dân et al.[26], which would improve the performance for unpopular torrents and thus the scalability of the system itself.
Bibliography


